digital

Migrating an Application from OpenVMS VAX to OpenVMS Alpha

OpenVMS

Migrating an Application from OpenVMS VAX to OpenVMS Alpha

Order Number: AA-QSBKA-TE

December 1995

This manual describes how to create an OpenVMS Alpha version of an OpenVMS VAX application.

Revision/Update Information: This is a new manual.

Software Version: OpenVMS Alpha Version 7.0
OpenVMS VAX Version 7.0

Digital Equipment Corporation
Maynard, Massachusetts

December 1995

The postpaid Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

ZK6459

This document is available on CD–ROM.

This document was prepared using VAX DOCUMENT Version 2.1.

Contents

Preface xi

1 Overview of the Migration Process

1.1 Compatibility of VAX and Alpha Systems 1–1
1.2 Differences Between the VAX and Alpha Architectures 1–4
1.2.1 User-Written Device Drivers 1–6
1.3 Migration Process 1–7
1.4 How to Assess the Portability of an Application 1–7
1.5 Migration Paths 1–8
1.6 Migration Support from Digital 1–10
1.6.1 Migration Assessment Service 1–10
1.6.2 Application Migration Detailed Analysis and Design Service 1–10
1.6.3 System Migration Detailed Analysis and Design Service 1–10
1.6.4 Application Migration Service 1–10
1.6.5 System Migration Service 1–10
1.7 Migration Training 1–11

Part I Planning for Migration

2 Selecting a Migration Method

2.1 Taking Inventory 2–1
2.2 How to Select a Migration Method 2–2
2.3 Which Migration Methods are Possible? 2–3
2.4 Coding Practices That Affect Recompilation 2–5
2.4.1 VAX MACRO Assembly Language 2–6
2.4.2 Privileged Code 2–6
2.4.3 Features Specific to the VAX Architecture 2–7
2.5 Identifying Dependencies on the VAX Architecture in Your Application 2–7
2.5.1 Performance Issues 2–9
2.5.1.1 Data Alignment 2–9
2.5.1.2 Data Types 2–10
2.5.2 Protection of Shared Data 2–11
2.5.2.1 Modifying Data in Memory 2–11
2.5.2.2 Reading or Writing Data Smaller Than a Quadword 2–12
2.5.2.3 Page Size Considerations 2–13
2.5.2.4 Order of Read/Write Operations on Multiprocessor Systems 2–14
2.5.3 Immediacy of Arithmetic Exception Reporting 2–15
2.5.4 Explicit Reliance on the VAX Procedure Calling Standard 2–16
2.5.5 Explicit Reliance on VAX Exception-Handling Mechanisms 2–16
2.5.5.1 Establishing a Dynamic Condition Handler 2–17
2.5.5.2 Accessing Data in the Signal and Mechanism Arrays 2–17
2.5.6 Modification of the VAX AST Parameter List 2–18

2.5.7 Explicit Dependency on the Form and Behavior of VAX Instructions 2–18
2.5.8 Generation of VAX Instructions at Run Time 2–18
2.6 Identifying Incompatibilities Between VAX and Alpha Systems 2–18
2.7 Deciding Whether to Recompile or Translate 2–20
2.7.1 Translating Your Application 2–23
2.7.2 Combining Native and Translated Images 2–24

3 Sample Migration Plan

3.1 Executive Summary 3–1
3.2 Technical Analysis 3–2
3.2.1 Application Characteristics 3–2
3.2.2 Software Architecture 3–2
3.2.3 Results of Image Analysis 3–3
3.2.4 Results of Source Analysis 3–4
3.3 Milestones and Deliverables 3–6
3.4 Technical Approach 3–6
3.4.1 Line Mode Prompt 3–6
3.4.2 Image Bridge 3–7
3.4.3 Floating-Point Format Decision 3–7
3.4.4 Full Omega-1 Exception Handling 3–7
3.4.5 Begin Code Generator Implementation 3–7
3.4.6 Build Applications 3–7
3.4.7 Test Code Generator 3–7
3.4.8 Test Complete Application 3–8
3.4.9 DECwindows Motif User Interface 3–8
3.4.10 Omega Quality Assurance and Field Test 3–8
3.5 Dependencies and Risks 3–8
3.6 Resource Requirements 3–9
3.6.1 Hardware 3–10
3.6.2 On-Site Training 3–10
3.6.3 Telephone Support 3–10
3.6.4 Testing Assistance 3–10
3.6.4.1 Testing the Code Generator 3–10
3.6.4.2 Testing Applications 3–10
3.6.4.3 Omega Quality Assurance 3–11

Part II Migrating the Application

4 Migrating Your Application

4.1 Setting Up the Migration Environment 4–1
4.1.1 Hardware 4–1
4.1.2 Software 4–2
4.2 Converting Your Application 4–3
4.2.1 Recompiling and Relinking 4–4
4.2.1.1 Native Alpha Compilers 4–4
4.2.1.2 VAX MACRO–32 Compiler for OpenVMS Alpha 4–5
4.2.1.3 Other Development Tools 4–6
4.2.2 Translating 4–6
4.2.2.1 VAX Environment Software Translator (VEST) and Translated Image Environment (TIE) 4–7
4.3 Debugging and Testing the Migrated Application 4–8

4.3.1 Debugging 4–8
4.3.1.1 Debugging with the OpenVMS Debugger 4–9
4.3.1.2 Debugging with the Delta Debugger 4–10
4.3.1.3 Debugging with the OpenVMS Alpha System-Code Debugger 4–10
4.3.2 Analyzing System Crashes 4–11
4.3.2.1 System Dump Analyzer 4–11
4.3.2.2 Crash Log Utility Extractor 4–12
4.3.3 Testing 4–12
4.3.3.1 VAX Tests 4–12
4.3.3.2 Alpha Tests 4–13
4.3.4 Uncovering Latent Bugs 4–13
4.4 Integrating the Migrated Application into a Software System 4–13

5 Recompiling and Relinking Overview

5.1 Overview 5–1
5.2 Recompiling Your Application with Native Alpha Compilers 5–1
5.3 Relinking Your Application on an Alpha System 5–2
5.4 Compatibility Between the Mathematics Libraries Available on VAX and Alpha Systems 5–4
5.5 Determining the Host Architecture 5–4

6 Adapting Applications to a Larger Page Size

6.1 Overview 6–1
6.1.1 Compatibility Features 6–1
6.1.2 Summary of Memory Management Routines with Potential Page-Size Dependencies 6–2
6.2 Examining Memory Allocation Routines 6–6
6.2.1 Allocating Memory in Expanded Virtual Address Space 6–6
6.2.2 Allocating Memory in Existing Virtual Address Space 6–8
6.2.3 Deleting Virtual Memory 6–9
6.3 Examining Memory Mapping Routines 6–10
6.3.1 Mapping into Expanded Virtual Address Space 6–10
6.3.2 Mapping a Single Page to a Specific Location 6–12
6.3.3 Mapping into a Defined Address Range 6–13
6.3.4 Mapping from an Offset into a Section File 6–19
6.4 Obtaining the Page Size at Run Time 6–20
6.5 Locking Memory in the Working Set 6–21

7 Preserving the Integrity of Shared Data

7.1 Overview 7–1
7.1.1 VAX Architectural Features That Guarantee Atomicity 7–2
7.1.2 Alpha Compatibility Features 7–3
7.2 Uncovering Atomicity Assumptions in Your Application 7–3
7.2.1 Protecting Explicitly Shared Data 7–5
7.2.2 Protecting Unintentionally Shared Data 7–8
7.3 Synchronizing Read/Write Operations 7–9
7.4 Ensuring Atomicity in Translated Images 7–10

8 Checking the Portability of Application Data Declarations

8.1	Overview	8–1
8.2	Checking for Dependence on a VAX Data Type	8–1
8.3	Examining Assumptions About Data-Type Selection	8–4
8.3.1	Effect of Data-Type Selection on Code Size	8–4
8.3.2	Effect of Data-Type Selection on Performance	8–4

9 Examining the Condition-Handling Code in Your Application

9.1	Overview	9–1
9.2	Establishing Dynamic Condition Handlers	9–1
9.3	Examining Condition-Handling Routines for Dependencies	9–2
9.4	Identifying Exception Conditions	9–6
9.4.1	Testing for Arithmetic Exceptions on Alpha Systems	9–8
9.4.2	Testing for Data-Alignment Traps	9–10
9.5	Performing Other Tasks Associated with Condition Handling	9–11

10 Translating Applications

10.1	DECmigrate for OpenVMS Alpha	10–1
10.2	DECmigrate: Translated Image Support	10–2
10.3	Translated Image Environment (TIE)	10–2
10.3.1	Problems and Restrictions	10–4
10.3.1.1	Condition Handler Restriction	10–4
10.3.1.2	Exception Handler Restrictions	10–4
10.3.1.3	Floating-Point Restrictions	10–4
10.3.1.4	Interoperability Restrictions	10–5
10.3.1.5	VAX C: Translated Program Restrictions	10–5
10.4	Translated Image Support	10–6
10.5	Translated Run-Time Libraries	10–10
10.5.1	CRF$FREE_VM and CRF$GET_VM: Translated Callers	10–11
10.6	Translated VAX C Run-Time Library	10–11
10.6.1	Problems and Restrictions	10–11
10.6.1.1	Functional Restrictions	10–11
10.6.1.2	Interoperability Restrictions	10–12
10.7	Translated VAX COBOL Programs	10–12
10.7.1	Problems and Restrictions	10–12

11 Ensuring Interoperability Between Native and Translated Images

11.1	Overview	11–1
11.1.1	Compiling Native Images That Can Interoperate with Translated Images	11–1
11.1.2	Linking Native Images That Can Interoperate with Translated Images	11–2
11.2	Creating a Native Image That Can Call a Translated Image	11–2
11.3	Creating a Native Image That Can Be Called by a Translated Image	11–5
11.3.1	Controlling Symbol Vector Layout	11–6
11.3.2	Creating Stub Images	11–8

Part III Layered Products

12 OpenVMS Alpha Compilers

12.1 Compatibility of DEC Ada Between Alpha Systems and VAX Systems . . . 12–1
12.1.1 Differences in Data Representation and Alignment . . . 12–2
12.1.2 Tasking Differences . . . 12–2
12.1.3 Differences in Language Pragmas . . . 12–2
12.1.4 Differences in the SYSTEM Package . . . 12–3
12.1.5 Differences Between Other Language Packages . . . 12–4
12.1.6 Changes to Predefined Instantiations . . . 12–4
12.2 Compatibility of DEC C for OpenVMS Alpha Systems with VAX C . . . 12–4
12.2.1 Language Modes . . . 12–4
12.2.2 DEC C for OpenVMS Alpha Systems Data-Type Mappings . . . 12–5
12.2.2.1 Specifying Floating-Point Mapping . . . 12–5
12.2.3 Built-in Functions That Access Alpha Instructions . . . 12–6
12.2.3.1 Accessing Alpha Instructions . . . 12–6
12.2.3.2 Accessing Alpha Privileged Architecture Library (PALcode) Instructions . . . 12–6
12.2.3.3 Ensuring the Atomicity of Combined Operations . . . 12–6
12.2.4 Differences Between the VAX C and DEC C for OpenVMS Alpha Systems Compilers . . . 12–7
12.2.4.1 Controlling Data Alignment . . . 12–7
12.2.4.2 Accessing Argument Lists . . . 12–7
12.2.4.3 Synchronizing Exceptions . . . 12–8
12.2.4.4 Dynamic Condition Handlers . . . 12–8
12.2.5 SYS$STARLET_C.TLB: Functionally Equivalent to STARLETSD.TLB . . . 12–8
12.2.6 VAX C Features Not Supported by /STANDARD=VAXC Mode . . . 12–9
12.3 Compatibility of DEC COBOL with VAX COBOL . . . 12–10
12.3.1 Command Line Qualifiers . . . 12–11
12.3.1.1 Qualifiers Shared by DEC COBOL and VAX COBOL . . . 12–11
12.3.1.2 DEC COBOL Qualifiers Not Available in VAX COBOL . . . 12–12
12.3.1.3 VAX COBOL Qualifiers Not Available in DEC COBOL . . . 12–13
12.3.2 Behavior Differences . . . 12–13
12.3.2.1 Specifying Alignment for Numeric Data Items with the DEC COBOL /ALIGNMENT Qualifier and Alignment Directives . . . 12–14
12.3.2.1.1 Using the /ALIGNMENT Qualifier . . . 12–14
12.3.2.1.2 Using Alignment Directives . . . 12–15
12.3.2.2 Validating Numeric Data with the DEC COBOL /CHECK=NODECIMAL Qualifier Option . . . 12–15
12.3.2.3 Converting Leading Blanks to Zeros with the DEC COBOL /CONVERT=LEADING_BLANKS Qualifier Option . . . 12–15
12.3.2.4 Specifying a Floating-Point Data Format with the DEC COBOL /FLOAT Qualifier . . . 12–16
12.3.2.5 Optimizing Your Code with the DEC COBOL /OPTIMIZE Qualifier . . . 12–16
12.3.2.6 Checking for Special Reserved Words with the DEC COBOL /RESERVED_WORDS Qualifier . . . 12–16
12.3.2.7 Calling Out Language Feature Extensions to the COBOL ANSI Standard with the DEC COBOL /STANDARD Qualifier . . . 12–17
12.3.2.7.1 /STANDARD=V3 Qualifier Option . . . 12–17
12.3.2.7.2 /STANDARD and /WARNINGS Qualifiers . . . 12–19

12.3.2.8 Calling Native and Translated Images with the DEC COBOL /TIE Qualifier . . . 12–20
12.3.2.9 VAX COBOL to DEC COBOL Program Conversion . . . 12–20
12.3.2.10 Program Structure . . . 12–20
12.3.2.11 COPY and REPLACE Statements . . . 12–21
12.3.2.12 MOVE Statement . . . 12–24
12.3.2.13 ACCEPT and DISPLAY Statements . . . 12–25
12.3.2.14 LINAGE Statement . . . 12–25
12.3.2.15 File Status Differences . . . 12–26
12.3.2.16 System Return Codes . . . 12–26
12.3.2.17 Storage Differences for Double-Precision Data Items . . . 12–27
12.3.2.18 RMS Special Registers . . . 12–27
12.4 Compatibility of Digital Fortran for OpenVMS Alpha with VAX FORTRAN . . . 12–28
12.4.1 Language Features . . . 12–28
12.4.1.1 Language Features Specific to Digital Fortran for OpenVMS Alpha . . . 12–29
12.4.1.2 Language Features Specific to DEC Fortran for OpenVMS VAX Systems . . . 12–30
12.4.1.3 Interpretation Differences . . . 12–31
12.4.2 Command Line Qualifiers . . . 12–32
12.4.2.1 Qualifiers Specific to Digital Fortran for OpenVMS Alpha . . . 12–32
12.4.2.2 Qualifiers Specific to DEC Fortran for OpenVMS VAX Systems . . . 12–33
12.4.3 Interoperability with Translated Shared Images . . . 12–34
12.4.4 Porting DEC Fortran for OpenVMS VAX Systems Data . . . 12–35
12.5 Compatibility of DEC Pascal for OpenVMS Alpha Systems with VAX Pascal . . . 12–35
12.5.1 New Features of DEC Pascal . . . 12–36
12.5.2 Establishing Dynamic Condition Handlers . . . 12–37
12.5.3 Modifying Default Alignment Rules for Record Fields . . . 12–37
12.5.4 Recommended Use of Predeclared Identifiers . . . 12–37
12.5.5 Platform-Dependent Features . . . 12–38
12.5.6 Obsolete Features . . . 12–38
12.5.6.1 /OLD_VERSION Qualifier . . . 12–38
12.5.6.2 /G_FLOATING Qualifier . . . 12–39
12.5.6.3 OVERLAID Attribute . . . 12–39

A Application Evaluation Checklist

Glossary

Index

Examples

5–1 Using the ARCH_TYPE Keyword to Determine Architecture Type . . . 5–5
6–1 Allocating Memory by Expanding Your Virtual Address Space . . . 6–8
6–2 Allocating Memory in Existing Address Space . . . 6–9
6–3 Mapping a Section into Expanded Virtual Address Space . . . 6–11
6–4 Mapping a Section into a Defined Area of Virtual Address Space . . . 6–15
6–5 Source Code Changes Required to Run Example 6–4 on an Alpha System . . . 6–17

6–6	Using the $GETSYI System Service to Obtain the CPU-Specific Page Size	6–20
7–1	Atomicity Assumptions in a Program with an AST Thread	7–5
7–2	Version of Example 7–1 with Synchronization Assumptions	7–7
8–1	Assumptions About Data Types in VAX C Code	8–3
9–1	Condition-Handling Routine	9–6
9–2	Sample Condition-Handling Program	9–13
11–1	Source Code for Main Program (MYMAIN.C)	11–3
11–2	Source Code for Shareable Image (MYMATH.C)	11–3

Figures

1–1	Methods for Moving VAX Applications to an Alpha System	1–9
2–1	Migrating a Program	2–4
3–1	Layer Structure of Omega-1	3–3
4–1	Migration Environments and Tools	4–3
6–1	Virtual Address Layout	6–7
6–2	Effect of Address Range on Mapping from an Offset	6–20
7–1	Synchronization Decision Tree	7–4
7–2	Atomicity Assumptions in Example 7–1	7–6
7–3	Order of Read and Write Operations on an Alpha System	7–10
8–1	Alignment of mystruct Using VAX C	8–6
8–2	Alignment of mystruct Using DEC C for OpenVMS Alpha Systems	8–6
9–1	32-Bit Signal Array on VAX and Alpha Systems	9–3
9–2	Mechanism Array on VAX and Alpha Systems	9–4
9–3	SS$_HPARITH Exception Signal Array	9–9
9–4	SS$_ALIGN Exception Signal Array	9–11

Tables

1–1	Comparison of Alpha and VAX Architectures	1–5
2–1	Floating-Point Data Type Support	2–7
2–2	Migration Path Comparison	2–20
2–3	Choice of Migration Method: Dealing with Architectural Dependencies	2–22
3–1	Image Analysis Results	3–4
3–2	Milestones and Deliverables	3–6
3–3	Omega Optional Product Dependencies	3–9
3–4	Summary of Digital Support	3–9
4–1	CLUE Differences Between OpenVMS VAX and OpenVMS Alpha	4–12
5–1	Linker Qualifiers and Options Specific to OpenVMS Alpha Systems	5–2
5–2	Linker Options Specific to OpenVMS VAX Systems	5–4
5–3	$GETSYI Item Codes That Specify Host Architecture	5–5
6–1	Potential Page-Size Dependencies in Memory Management Routines	6–2
6–2	Potential Page-Size Dependencies in Run-Time Library Routines	6–6
8–1	Comparison of VAX and Alpha Native Data Types	8–2

9–1	Architecture-Specific Hardware Exceptions	9–7
9–2	Exception Summary Argument Fields	9–9
9–3	Run-Time Library Condition-Handling Support Routines	9–12
10–1	Support for Translated Images on OpenVMS Alpha Versions	10–2
10–2	Interoperability Documentation	10–3
10–3	Run-Time Library Logical Names	10–11
12–1	Modes of Operation of the DEC C for OpenVMS Alpha Systems	12–4
12–2	Arithmetic Data-Type Sizes in DEC C for OpenVMS Alpha Compiler	12–5
12–3	DEC C Floating-Point Mappings	12–6
12–4	DEC C Compiler Features Specific to Alpha Systems	12–6
12–5	Atomicity Built-Ins	12–7
12–6	Qualifiers and Options Shared by DEC COBOL and VAX COBOL	12–11
12–7	DEC COBOL Qualifiers Not Available in VAX COBOL	12–12
12–8	VAX COBOL Qualifiers Not Available in DEC COBOL	12–13
12–9	I/O File Status Codes for the /STANDARD Qualifier	12–18
12–10	Digital Fortran for OpenVMS Alpha Qualifiers Not in DEC Fortran for OpenVMS VAX Systems	12–32
12–11	DEC Fortran for OpenVMS VAX Systems Qualifiers Not in Digital Fortran for OpenVMS Alpha	12–34
12–12	Floating-Point Data on VAX and Alpha Systems	12–35
12–13	New Features of DEC Pascal	12–36
12–14	Recommended Use of Predeclared Identifiers	12–37

Preface

Migrating an Application from OpenVMS VAX to OpenVMS Alpha is designed to assist developers in moving OpenVMS VAX applications to an OpenVMS Alpha system or a Mixed-Architecture cluster. The manual consists of the following chapters:

- Chapter 1 provides an overview of the relationship of OpenVMS and the VAX and Alpha architectures, and of the process of migrating an application from a VAX to an Alpha system. It includes information on the following:
  - Areas in which OpenVMS Alpha is highly compatible with OpenVMS VAX
  - Comparison of the Alpha architecture with other RISC architectures and with the VAX architecture
  - Overview of the stages in the migration process
  - The two main migration paths—recompiling source code and translating VAX images
  - Migration support available from Digital
- Chapter 2 considers the differences between the two main migration paths and the issues involved in choosing which path to take in migrating your application. It also describes how to analyze the individual parts of your application to identify architectural differences that affect migration and how to assess what is involved in resolving those differences.
- Chapter 3 contains a sample migration plan.
- Chapter 4 describes the steps in the actual migration, from setting up your migration environment to integrating the migrated application into a new environment.
- Chapter 5 provides an overview of converting your application by recompiling and relinking.
- Chapter 6 describes how to handle dependencies your application may have on the VAX page size.
- Chapter 7 describes how to handle dependencies your application may have on the synchronization provided by the VAX architecture with regard to data access by multiple processes.
- Chapter 8 describes the implications of data declarations on an Alpha system, including alignment concerns.
- Chapter 9 describes how to handle dependencies your application may contain on the VAX condition-handling facility.
- Chapter 10 discusses translating VAX images to run on Alpha systems.

- Chapter 11 describes how to create native Alpha images that can call and be called by translated VAX images.
- Chapter 12 contains brief summaries of the new and changed features supported by the Ada, C, COBOL, FORTRAN, and Pascal programming languages on Alpha systems.
- Appendix A contains a checklist that you can use to evaluate your application for migration from OpenVMS VAX to OpenVMS Alpha.

Intended Audience

This manual is intended for experienced software engineers responsible for migrating application code written in high- or mid-level programming languages.

Related Documents

This manual is part of a set of manuals that describes various aspects of migrating from OpenVMS VAX to OpenVMS Alpha systems. The other manuals in this set are as follows:

- *Migrating an Environment from OpenVMS VAX to OpenVMS Alpha* describes how to migrate a computing environment from an OpenVMS VAX system to an OpenVMS Alpha system or a Mixed-Architecture Cluster. It provides an overview of the VAX to Alpha migration process and describes the differences in system and network management on VAX and Alpha computers.
- *Porting VAX MACRO Code from OpenVMS VAX to OpenVMS Alpha* describes how to port VAX MACRO code to an Alpha system using the MACRO–32 compiler for OpenVMS Alpha. It describes the features of the compiler, presents a methodology for porting VAX MACRO code, identifies nonportable coding practices, and recommends alternatives to such practices. The manual also provides a reference section with detailed descriptions of the compiler's qualifiers, directives, and built-ins, and the system macros created for porting to Alpha systems.

In addition, the *DECmigrate for OpenVMS AXP Systems Translating Images* manual describes the VAX Environment Software Translator (VEST) utility. This manual is distributed with the optional layered product, DECmigrate for OpenVMS Alpha, which supports the migration of OpenVMS VAX applications to OpenVMS Alpha systems. The manual describes how to use VEST to convert most user-mode VAX images to translated images that can run on Alpha systems; how to improve the run-time performance of translated images; how to use VEST to trace Alpha incompatibilities in a VAX image back to the original source files; and how to use VEST to support compatibility among native and translated run-time libraries. The manual also includes complete VEST command reference information.

For additional information on OpenVMS products and services, access the Digital OpenVMS World Wide Web site. Use the following URL:

```
http://www.openvms.digital.com
```

Reader's Comments

Digital welcomes your comments on this manual.

Print or edit the online form SYS$HELP:OPENVMSDOC_COMMENTS.TXT and send us your comments by:

Internet	**openvmsdoc@zko.mts.dec.com**
Fax	603 881-0120, Attention: OpenVMS Documentation, ZK03-4/U08
Mail	OpenVMS Documentation Group, ZKO3-4/U08 110 Spit Brook Rd. Nashua, NH 03062-2698

How To Order Additional Documentation

Use the following table to order additional documentation or information. If you need help deciding which documentation best meets your needs, call 800-DIGITAL (800-344-4825).

Telephone and Direct Mail Orders

Location	Call	Fax	Write
U.S.A.	DECdirect 800–DIGITAL 800–344–4825	Fax: 800–234–2298	Digital Equipment Corporation P.O. Box CS2008 Nashua, NH 03061
Puerto Rico	809–781–0505	Fax: 809–749–8300	Digital Equipment Caribbean, Inc. 3 Digital Plaza, 1st Street, Suite 200 P.O. Box 11038 Metro Office Park San Juan, Puerto Rico 00910–2138
Canada	800–267–6215	Fax: 613–592–1946	Digital Equipment of Canada, Ltd. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6 Attn: DECdirect Sales
International	—	—	Local Digital subsidiary or approved distributor
Internal Orders	DTN: 264–4446 603–884–4446	Fax: 603–884–3960	U.S. Software Supply Business Digital Equipment Corporation 10 Cotton Road Nashua, NH 03063–1260

ZK–7654A–GE

Conventions

The name of the OpenVMS AXP operating system has been changed to OpenVMS Alpha. Any references to OpenVMS AXP or AXP are synonymous with OpenVMS Alpha or Alpha.

In this manual, every use of DECwindows and DECwindows Motif refers to DECwindows Motif for OpenVMS software.

The following conventions are also used in this manual:

Ctrl/*x*	A sequence such as Ctrl/*x* indicates that you must hold down the key labeled Ctrl while you press another key or a pointing device button.
Return	In examples, a key name enclosed in a box indicates that you press a key on the keyboard. (In text, a key name is not enclosed in a box.)
. . .	Horizontal ellipsis points in examples indicate one of the following possibilities: • Additional optional arguments in a statement have been omitted. • The preceding item or items can be repeated one or more times. • Additional parameters, values, or other information can be entered.
. . .	Vertical ellipsis points indicate the omission of items from a code example or command format; the items are omitted because they are not important to the topic being discussed.
( )	In command format descriptions, parentheses indicate that, if you choose more than one option, you must enclose the choices in parentheses.
[ ]	In command format descriptions, brackets indicate optional elements. You can choose one, none, or all of the choices. (Brackets are not optional, however, in the syntax of a directory name in an OpenVMS file specification or in the syntax of a substring specification in an assignment statement.)
{ }	In command format descriptions, braces indicate a required choice of options; you must choose one of the options listed.
boldface text	Boldface text represents the introduction of a new term or the name of an argument, an attribute, or a reason. Boldface text is also used to show user input in Bookreader versions of the book.
italic text	Italic text emphasizes important information, complete titles of manuals, or variables. Variables include information that varies in system output (Internal error *number*), in command lines (/PRODUCER=*name*), and in command parameters in text (where *device-name* contains up to five alphanumeric characters).
UPPERCASE TEXT	Uppercase text indicates a command, the name of a routine, the name of a file, or the abbreviation for a system privilege.
`Monospace type`	Monospace type indicates code examples and interactive screen displays. In the C programming language, monospace type in text identifies the following elements: keywords, the names of independently compiled external functions and files, syntax summaries, and references to variables or identifiers introduced in an example.
-	A hyphen at the end of a command format description, command line, or code line indicates that the command or statement continues on the following line.
numbers	All numbers in text are assumed to be decimal unless otherwise noted. Nondecimal radixes—binary, octal, or hexadecimal—are explicitly indicated.

1
Overview of the Migration Process

For many applications, migrating from OpenVMS VAX to OpenVMS Alpha is straightforward. If your application runs only in user mode and is written in a standard high-level language, you most likely can recompile it with a native Alpha compiler and relink it to produce a version that runs successfully on an Alpha system. This book is intended to help you evaluate your application and to handle the relatively few cases that are more complicated.

1.1 Compatibility of VAX and Alpha Systems

The OpenVMS Alpha operating system is designed to preserve as much compatibility with the OpenVMS VAX user, system management, and programming environments as possible. For general users and system managers, OpenVMS Alpha has the same interfaces as OpenVMS VAX. For programmers, the goal is to come as close as possible to a "recompile, relink, and run" model for migration.

Many aspects of an application running on an OpenVMS VAX system remain unchanged on an Alpha system:

User Interface

- **DIGITAL Command Language (DCL)**

  The DIGITAL Command Language (DCL), the standard user interface to OpenVMS, remains essentially unchanged with OpenVMS Alpha. All commands, qualifiers, and lexical functions available on OpenVMS VAX also work on OpenVMS Alpha.

- **Command Procedures**

  Command procedures written for earlier versions of OpenVMS VAX continue to work on an Alpha system without change. However, certain command procedures, such as build procedures, must be changed to accommodate new compiler qualifiers and linker switches. Linker options files will also require modification, especially for shareable images.

- **DECwindows**

  The window interface, DECwindows Motif, is unchanged.

- **DECforms**

  The DECforms interface is unchanged.

- **Editors**

  The two standard OpenVMS editors, EVE and EDT, are unchanged.

System Management Interface

The system management utilities are mostly unchanged. One major exception is that device configuration functions, which appear in the System Generation utility (SYSGEN) on VAX systems, are provided in the System Management utility (SYSMAN) for OpenVMS Alpha. For more information, see *A Comparison of System Management on OpenVMS AXP and OpenVMS VAX.*

Programming Interface

In general, the system service and run-time library (RTL) calling interfaces remain unchanged.[1] You do not need to change the definitions of arguments. The few differences fall into two categories:

- Some system services and RTL routines (such as the memory management system and exception-handling services) operate somewhat differently on VAX and Alpha systems. See the *OpenVMS System Services Reference Manual* and the Bookreader version of the *OpenVMS RTL Library (LIB$) Manual* for further information.
- A few RTL routines are so closely tied to the VAX architecture that their presence on an Alpha system would not be meaningful:

Routine Name	Restriction
LIB$DECODE_FAULT	Decodes VAX instructions.
LIB$DEC_OVER	Applies to VAX Processor Status Longword (PSL) only.
LIB$ESTABLISH	Similar functionality supported by compilers on Alpha systems.
LIB$FIXUP_FLT	Applies to VAX PSL only.
LIB$FLT_UNDER	Applies to VAX PSL only.
LIB$INT_OVER	Applies to VAX PSL only.
LIB$REVERT	Supported by compilers on Alpha systems.
LIB$SIM_TRAP	Applies to VAX code.
LIB$TPARSE	Requires action routine interface changes. Replaced by LIB$TABLE_PARSE.

Most VAX images that call these services and routines will work when translated and run under the Translated Image Environment (TIE) on OpenVMS Alpha. For more information on TIE, see Section 4.2.2.1 and *DECmigrate for OpenVMS AXP Systems Translating Images*.

Data

The on-disk format for ODS-2 data files is the same on VAX and Alpha systems. However, ODS-1 files are not supported on OpenVMS Alpha.

Record Management Services (RMS) and file management interfaces are unchanged.

The IEEE little-endian data types S_floating and T_floating have been added.

[1] Effective with Version 7.0, OpenVMS Alpha provides many system services and RTL routines to support 64-bit addressing. Since these are not available on VAX systems and are therefore not a VAX-to-Alpha migration issue, they are not discussed in this manual.

Most VAX data types are retained in the Alpha architecture; however, support for H_floating and full-precision D_floating has been eliminated from hardware to improve overall system performance.

Alpha hardware converts D_floating data to G_floating for processing. On VAX systems, D_floating has 56 fraction bits (D56) and 16 decimal digits of precision. On Alpha systems, D_floating has 53 fraction bits (D53) and 15 decimal digits of precision.

The H_floating and D_floating data types can usually be replaced by G_floating or one of the IEEE formats. However, if you require H_floating or the extra precision of D56 (56-bit D_floating), you may have to translate part of your application.

Databases

Standard Digital databases (such as Oracle Rdb) function the same on VAX and Alpha systems.

Network Interfaces

VAX and Alpha systems both support the following interfaces:

- Interconnects
  - Ethernet
  - X.25
  - FDDI
- Protocols
  - DECnet (Phase IV in Version 7.0; Phase V in the optional DECnet/OSI kit)
  - TCP/IP
  - OSI
  - LAD/LAST
  - LAT (Local Area Transport)
- Peripheral connections
  - TURBOchannel
  - SCSI
  - Ethernet
  - CI
  - DSSI
  - XMI
  - Futurebus/Plus
  - VME

1.2 Differences Between the VAX and Alpha Architectures

The VAX architecture is a robust, flexible, complex instruction set computer (CISC) architecture used across the entire family of VAX systems. The use of a single, integrated VAX architecture with the OpenVMS operating system permits an application to be developed on a VAXstation, prototyped on a small VAX system, and put into production on a large VAX processor or run on a fault-tolerant VAXft processor. The advantage of the VAX system approach is that it enables individual solutions to be tailored and fitted easily into a larger, enterprisewide solution. The hardware design of VAX processors is particularly suitable for high-availability applications, such as dependable applications for mission-critical business operations and server applications for a wide variety of distributed client/server environments.

The Alpha architecture implemented by Digital is a high-performance reduced instruction set computing (RISC) architecture that can provide 64-bit processing on a single chip. It processes 64-bit virtual and physical addresses and 64-bit integers and floating-point numbers. The 64-bit capability is especially useful for applications that require high-performance and very large addressing capacity. For example, Alpha processors are especially appropriate for graphics or numeric-intensive software applications such as econometric or weather forecasting that involve imaging, multimedia, visualization, simulation, and modeling.

The Alpha architecture is designed to be scalable and open. It can be implemented on a single chip in a palmtop system or with thousands of chips in a massively parallel supercomputer. The architecture also supports multiple operating systems, including OpenVMS Alpha.

Table 1–1 summarizes some major differences between the Alpha and VAX architectures.

Table 1–1 Comparison of Alpha and VAX Architectures

Alpha	VAX
• 64-bit addresses	• 32-bit addresses
• 64-bit processing	• 32-bit processing
• Multiple operating systems: OpenVMS, Digital UNIX, Windows NT	• One operating system: OpenVMS
• Instructions – Simple – All same length (32 bits)	• Instructions – Some complex – Variable length
• Load/store memory access	• Permits combining operations and memory access in a single instruction
• Severe penalty for unaligned data	• Moderate penalty for unaligned data
• Many registers	• Relatively few registers
• Out-of-order instruction completion	• Instructions completed in order issued
• Deep pipelines and branch prediction	• Limited use of pipelines
• Large page size (which varies from 8 KB to 64 KB, depending on hardware)	• Smaller page size (512 bytes)

General RISC Characteristics

Some features of the Alpha architecture are typical of newer RISC architectures in general. The following features are especially important:

- A simplified instruction set

  The Alpha architecture uses relatively simple instructions, all of which are 32 bits long. Common instructions require only one clock cycle. Uniformly sized simple instructions allow a RISC implementation to achieve high performance goals by adopting techniques such as **multiple instruction issue** and optimized instruction scheduling.

- Multiple instruction issue

  The earliest Alpha platform issued two instructions per clock cycle. Current machines (EV5 or higher) issue four instructions per clock cycle.

- A load/store operation model

  The Alpha architecture defines 32 64-bit integer registers and 32 64-bit floating-point registers. Most data manipulation occurs between registers. Typically, operands are loaded from memory into registers before an operation; after the operation, the results are stored in memory from a result register.

  Restricting operations to register operands allows the use of a simple, uniform instruction set. Moreover, the separation of memory access from arithmetic operations results in a large performance gain in a system that can fully exploit pipelining, instruction scheduling, and parallel operational units.

- Elimination of microcode

  Because the Alpha architecture does not use microcode, Alpha processors are saved the time required to fetch microcode instructions from random-access memory (RAM) in order to execute a machine instruction.

- Out-of-order completion of instructions

  The Alpha architecture does not require that instructions always complete in the order in which they are issued. As a result, an Alpha processor can improve performance by delaying the reporting of an arithmetic or floating-point exception until the execution stream allows the reporting without a performance penalty.

Alpha Specific Characteristics

Besides these generic RISC characteristics, the Alpha architecture offers features that promote running migrated VAX applications on an Alpha system. These features include:

- Hardware support for all VAX data types except packed decimal, H_floating, and D_floating. (For information on what to do if your application uses H_floating or D_floating data, see Section 2.5.1.2.)
- Certain privileged architecture features, such as four processor modes (user, supervisor, executive, and kernel), 32 interrupt priority levels (IPLs), and asynchronous system traps (ASTs).
- A privileged architecture library (PAL), part of an environment known as PALcode, that supports the atomic execution of certain VAX operations, such as Change Mode (CHM*x*), Probe (PROBE*x*), queue instructions, and REI.

  The Alpha architecture does not favor a particular operating system. To accommodate different operating systems, it enables the creation of privileged architecture library code (PALcode).

  Furthermore, certain OpenVMS Alpha compilers, such as C and the MACRO–32 compiler, provide PALcode built-ins that supplement the instructions available in the Alpha instruction set. For example, the MACRO–32 compiler provides built-ins that emulate those VAX instructions for which there are no Alpha equivalents and a built-in that enables you to write your own PALcode.

  PALcode can be used to access internal hardware registers and physical memory. PALcode can provide direct correspondence of physical and virtual memory. For more information about PALcode, see the *Alpha Architecture Reference Manual*.

1.2.1 User-Written Device Drivers

Formal support for user-written device drivers and a new interface known as the Step 2 driver interface were introduced in OpenVMS AXP Version 6.1. The Step 2 driver interface supports user-written device drivers in the C programming language (as well as MACRO and BLISS). It replaced the temporary Step 1 driver interface that was provided in OpenVMS Alpha Versions 1.0 and 1.5.

There is no formal support for writing OpenVMS VAX device drivers in C. For example, OpenVMS VAX does not provide .h files for internal OpenVMS (lib) data structures.

The Step 2 driver interface has increased the differences between OpenVMS Alpha and OpenVMS VAX device drivers. Device driver source files written in VAX MACRO or BLISS can be kept common between OpenVMS Alpha and VAX through the use of conditional compilation and user-written macros.

The advisability of this approach depends greatly on the nature of the individual driver.[2] It is likely that in future versions of OpenVMS Alpha, the I/O subsystem will continue to evolve in directions that will have an impact on device drivers. This could increase the differences between OpenVMS Alpha and VAX device drivers and add more complexity to common driver sources. For this reason, a fully common driver source file approach might not be advisable for the long term.

Depending on the individual driver, it might be advisable to partition the driver into a common module and an architecture-specific one. For example, if one were writing a device driver that does disk compression, then the compression algorithm could readily be isolated into an architecture independent module. One could also avoid operating-system-specific data structures in such common modules with the intent of having some common modules across various types of operating systems; for example, OpenVMS, Windows NT, and Digital UNIX.

For more information about writing OpenVMS Alpha device drivers in C, see the *OpenVMS Alpha Device Support: Developer's Guide*.

1.3 Migration Process

The process for converting your VAX programs to run on an Alpha system includes the following stages:

1. Evaluate the code to be migrated:
   - Take inventory of the elements of your application and its environment. Identify any dependencies on other programs.
   - Review code in each element to find potential obstacles to migration.
   - Decide on the best method for moving each part of the application to the Alpha system.
2. Write a migration plan.
3. Set up the migration environment.
4. Migrate your application.
5. Debug and test the migrated application.
6. Integrate the migrated software into a software system.

There are a number of tools and Digital services available to help you migrate your applications to OpenVMS Alpha. These tools are described in the context of the process described in this manual. The migration services are summarized in Section 1.6.

1.4 How to Assess the Portability of an Application

The portability of an application depends on the language in which it is written, the amount of nonstandard code it contains, the number of architectural dependencies it contains, and whether a compiler is available for the language in which the application is written. While it is possible to introduce architectural dependencies in applications written in high-level languages, they are more likely to occur in applications written in mid- and low-level languages.

---

[2] With OpenVMS Version 7.0, the difference is even greater due to the 64-bit support.

In general, if your application is written in a high-level programming language, you should be able to run it on an Alpha system with a minimum amount of effort. High-level languages insulate applications from dependence on the underlying machine architecture, and, for the most part, the programming environment on Alpha systems duplicates the programming environment on VAX systems. Using native Alpha versions of the language compilers and the Linker utility (linker), you can recompile and relink the source files that make up your application to produce a native Alpha image.

If your application is written in VAX MACRO, you may be able to run it on an Alpha system with a minimum amount of effort, although it is more likely to contain some dependencies on the underlying VAX architecture, some of which may require your intervention.

Privileged applications, which run in inner modes or at elevated interrupt priority levels (IPLs), may require significant changes because of assumptions incorporated in the code about the internal operation of the operating system. Typically, such applications also require significant changes after a major release of the OpenVMS VAX operating system.

Recently, Digital introduced new versions of several compilers. It is likely that the applications that you want to move to an OpenVMS Alpha system were compiled using the earlier VAX compilers.

To assess the portability of an application, consider the following:

- The application's dependencies on the VAX architecture
- The differences between the VAX and Alpha language compilers

1.5 Migration Paths

There are two ways to convert a program to run on an Alpha system:

- Recompiling and relinking, which creates native Alpha images
- Translating, which creates native Alpha images with some routines emulated under TIE

These two methods are shown in Figure 1–1. Section 2.2 discusses factors to consider when choosing a migration method.

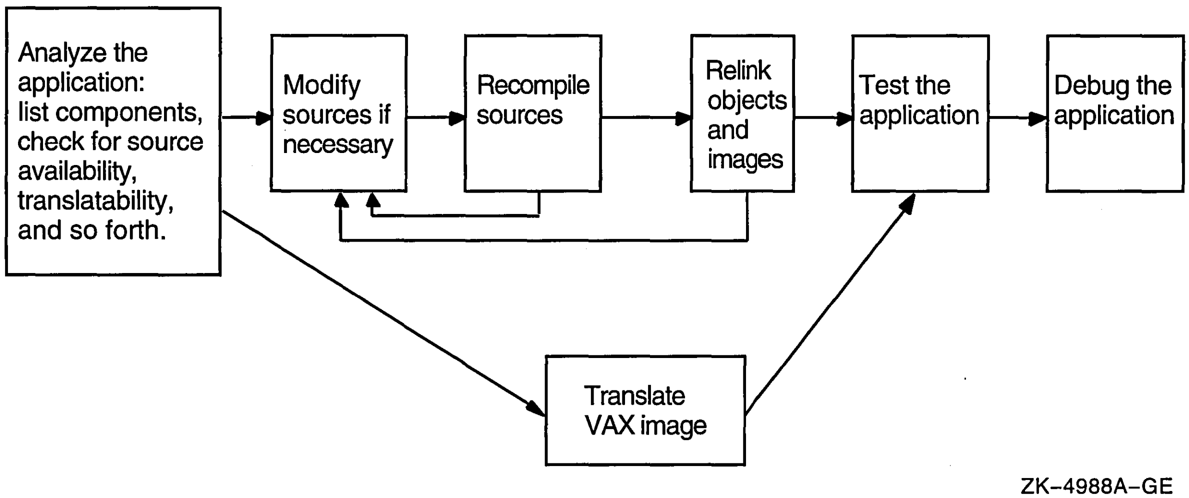
Recompiling and Relinking

The most effective way to convert a program from OpenVMS VAX to OpenVMS Alpha is to recompile the source code using a native Alpha compiler (such as DEC C or DEC Fortran) and then to relink the resulting object files and any required shareable images with the OpenVMS Linker. This method produces a native Alpha image that takes full advantage of the speed of the Alpha system.

Translating

Despite differences between VAX and Alpha systems, you can run most user-mode VAX images without error on an Alpha system by using the VAX Environment Software Translator (VEST), which is part of the DECmigrate for OpenVMS Alpha layered product. For a list of exceptions, see Section 2.3. This process provides a higher degree of VAX compatibility than recompiling the sources, but since the translated image does not provide the same high performance as a recompiled image, translation is used primarily as a safety net when recompiling

Figure 1–1 Methods for Moving VAX Applications to an Alpha System



is impossible or impractical. For example, translation is used in the following situations:

- When an appropriate compiler is not yet available for OpenVMS Alpha
- When source files are not available

VEST translates the VAX binary image file into a native Alpha image that runs under the Translated Image Environment (TIE) on an Alpha system. (TIE is a shareable image that is part of OpenVMS Alpha.) Translation does not involve running a VAX image under emulation or interpretation (with certain limited exceptions). Instead, the new Alpha image contains Alpha instructions that perform operations identical to those performed by the instructions in the original VAX image.

A translated image should run as fast on an Alpha system as the original image runs on a VAX system. However, since the translated image does not benefit from the optimizing compilers that take full advantage of the Alpha architecture, it will typically run only about 25 to 40 percent as fast as a native Alpha image. Major causes of this reduced performance are unaligned data and extensive use of complex VAX instructions.

For more information on image translation and VEST, see Section 4.2.2.1 and *DECmigrate for OpenVMS AXP Systems Translating Images*.

Mixing Native Alpha and Translated Images

You can mix migration methods among the individual images that comprise an application. An application can also be partially translated as one stage in a migration: this allows the application to run and to be tested on Alpha hardware before being completely recompiled. For more information about interoperability of native Alpha and translated VAX images within an application, see Section 2.7.2.

1.6 Migration Support from Digital

Digital offers a variety of services to help you migrate your applications to OpenVMS Alpha.

Digital customizes the level of service to meet your needs. The VAX to Alpha migration services available include the following:

- Migration Assessment
- Application Migration Detailed Analysis and Design
- System Migration Detailed Analysis and Design
- Application Migration
- System Migration

To determine which services are appropriate for you, contact your Digital account representative or authorized reseller, or the Digital Systems integration Business Development Manager for your region (AP, Americas, Europe). Call 800-832-6277 (within the United States) or 603-884-8990 (outside the United States).

1.6.1 Migration Assessment Service

The Migration Assessment service assesses the VAX system and application environment to be migrated to the Alpha platform. The objectives of the migration are reviewed and a complete current state configuration is completed. The desired end state is determined and risks and constraints are identified. Finally, several migration scenarios are developed.

1.6.2 Application Migration Detailed Analysis and Design Service

The Application Migration Detailed Analysis and Design service does a detailed analysis of an in-house developed application, creating a report of all VAX dependencies within all modules and recommendations as to what modifications should be made to migrate the application to Alpha. Acceptance criteria is specified for performance and functionality.

1.6.3 System Migration Detailed Analysis and Design Service

The System Migration Detailed Analysis and Design service performs a detailed analysis of the current system environment which includes hardware, software (Digital and third party, excluding in-house developed applications) and network components. The best tools and migration methods are determined and a project plan, which maps the steps from the current to the future state, is created.

1.6.4 Application Migration Service

The Application Migration service migrates an in-house developed application from an OpenVMS VAX platform to an Alpha platform. Each code module is either recompiled or translated depending on source code availability. VAX dependencies are removed beforehand. Finally the entire application is relinked and tested on the Alpha platform. The application is then deployed on the target system(s).

1.6.5 System Migration Service

The System Migration service migrates an OpenVMS system (single node or cluster) from the VAX platform to the Alpha platform. The customer's system availability and performance requirements are reviewed and acceptance testing methodology and criteria are determined.

1.7 Migration Training

Digital Customer Training offers several seminars and courses to provide migration training to third-party application developers and end users. The first course in the following list is designed for technical or MIS managers, and the others are designed for experienced OpenVMS VAX programmers:

- *Alpha Planning Seminar*—2 days
- *Migrating HLL Applications to OpenVMS Alpha*—3 days
- *Migrating MACRO–32 Applications to OpenVMS Alpha*–2 days

To obtain a schedule and enrollment information in the United States, call 800-332-5656. In other locations, contact your Digital account representative or authorized reseller.

Part I

Planning for Migration

2 Selecting a Migration Method

Evaluating your application identifies the work to be done and allows you to plan the rest of the migration.

The evaluation process has three main stages:

1. General inventory, including identifying dependencies on other software
2. Source analysis to identify coding practices that affect migration
3. Selection of a migration method: rebuilding from source code or translating

When you have completed these steps, you will have the information necessary to write an effective migration plan.

2.1 Taking Inventory

The first step in evaluating an application for migration is to determine exactly what has to be migrated. This includes not only the application itself, but everything that the application requires in order to run properly. To begin evaluating your application, identify and locate the following items:

- Parts of the application
  - Source modules for the main program
  - Shareable images
  - Object modules
  - Libraries (object module, shareable image, text, or macro)
  - Data files and databases
  - Message files
  - CLD files
  - UIL and UID files for DECwindows support
- Other software on which your application depends, for example:
  - Run-time libraries
  - Digital layered products
  - Third-party products

  To help identify dependencies on other code, use VEST with the qualifier /DEPENDENCY. VEST/DEPENDENCY identifies executable and shareable images on which your application depends, such as run-time libraries, system services, and other applications. For details on using VEST/DEPENDENCY, see *DECmigrate for OpenVMS AXP Systems Translating Images*.

- Required operating environment
  - System characteristics

    What sort of system is required to run and maintain your application; for example, how much memory is required, how much disk space, and so on?
  - Build procedures

    This includes Digital tools such as Code Management System (CMS) and Module Management System (MMS).
  - Testing suite

    You will need your tests to confirm that the migrated application runs correctly and to evaluate its performance.

Many of these items have already been migrated to OpenVMS Alpha, for example:

- Digital software bundled with OpenVMS
  - RTLs
  - Other shareable libraries, such as those supplying callable utility routines and application library routines
- Digital layered products
  - Compilers and compiler RTLs
  - Database managers
  - Networking environment
- Third-party products

  Many third-party applications now run on OpenVMS Alpha. To determine whether a particular application has been migrated, contact the application vendor.

You will be responsible for migrating your application and your development environment, including build procedures and testing suites.

2.2 How to Select a Migration Method

When you have completed the inventory of your application, you must decide how to migrate each part of it: by recompiling and relinking or by translating. The large majority of applications can be migrated just by recompiling and relinking them. If your application runs only in user mode and is written in a standard high-level language, it is probably in this category. For the major exceptions, see Section 2.4.

The remainder of this chapter discusses how to choose a migration method for the relatively few applications that require more work to migrate. To make this decision, you will need to know which methods are possible for each part of the application, and how much work will be required for each method.

Note

The following process assumes that you will recompile your application if possible, and use translation only for parts that cannot be recompiled or as a temporary measure in the course of your migration.

The following sections outline a process for choosing a migration method. This process includes the following steps:

1. Determine which of the two migration methods is possible.

   Under most conditions, you can either recompile and relink your program or translate the VAX image. Section 2.3 describes cases where only one migration method is available.

2. Identify architectural dependencies that affect recompilation.

   Even if your application is generally suitable to be recompiled, it may contain code that depends on features of the VAX architecture that are incompatible with the Alpha architecture.

   Section 2.4 discusses these dependencies and provides information that allows you to identify them and to begin to estimate the type and amount of work required to accommodate any dependencies you find.

   Section 2.6 describes tools and methods you can use to help answer the questions that come up in evaluating your application.

3. Decide whether to recompile or translate.

   After you have evaluated your application, you must decide which migration method to use. Section 2.7 describes how to make the decision by balancing the advantages and disadvantages of each method.

   If you cannot recompile and relink your program, or if the VAX image uses features specific to the VAX architecture, you may wish to translate that image. Section 2.7.1 describes ways to increase the compatibility and performance of translated images.

As shown in Figure 2–1, the evaluation process consists of a series of questions and some tasks you can perform to help answer those questions. Digital provides a number of tools that you can use to help answer the questions; these tools are described at the relevant points in the process.

2.3 Which Migration Methods are Possible?

In most cases, you can either recompile and relink, or translate your application. However, depending on the design of your application, only one of the two migration paths may be available to you:

- Programs that cannot be recompiled

  The following types of images must be translated:

  – Software that is written in a programming language for which no Alpha compiler is yet available, for example VAXscan

  – Executable and shareable images for which the source code is not available

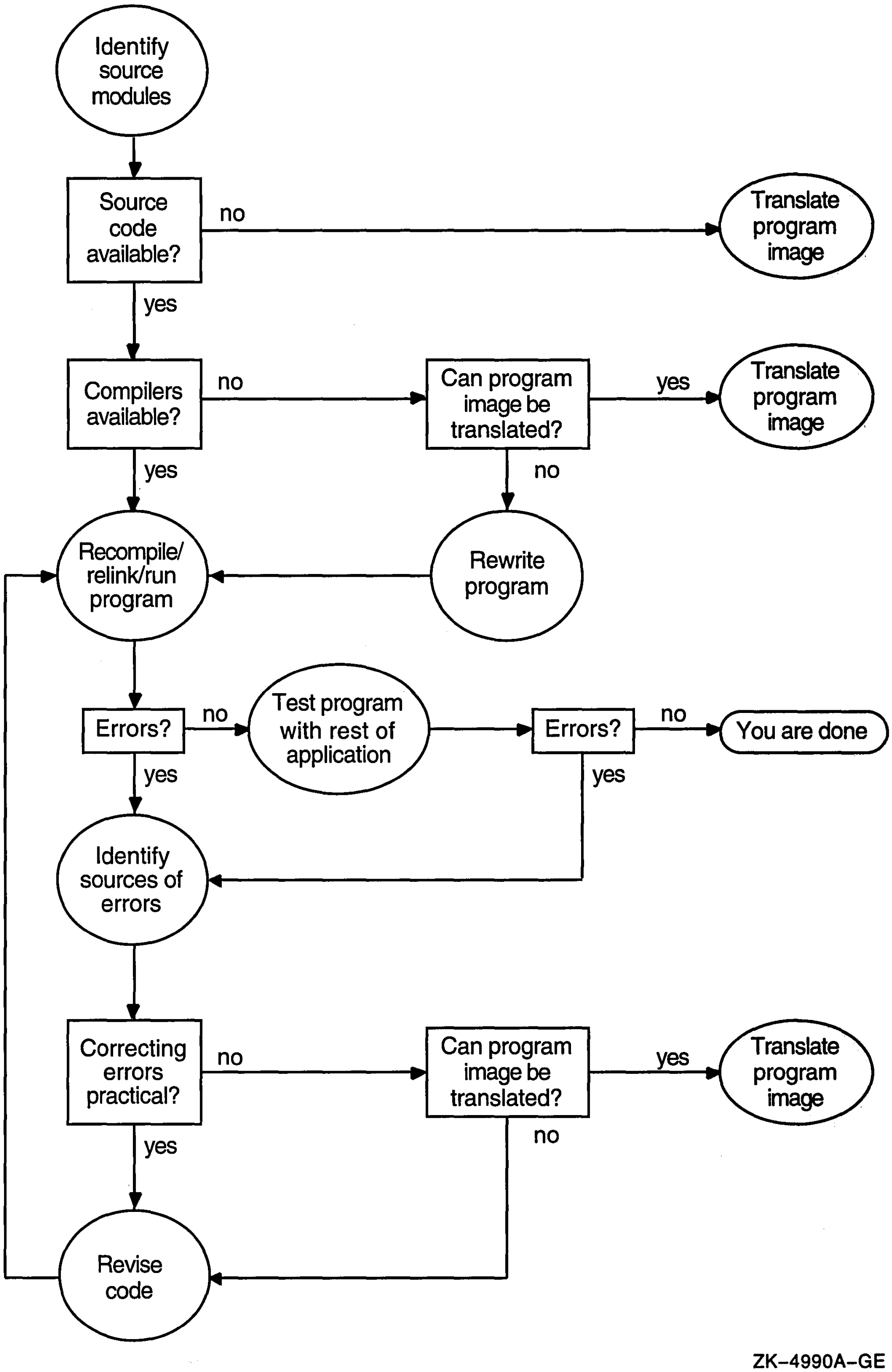
  – Programs that require H_floating or 56-bit D_floating data

- Images that cannot be translated

  The source code must be recompiled and relinked (and possibly revised) for the following types of images:

  – Images produced prior to OpenVMS VAX Version 4.0

Figure 2–1 Migrating a Program



- Images written in Ada
- Images that call or are called by images written in Ada
- Images that use PDP–11 compatibility mode
- Based images
- Images that contain coding practices that are currently unsupported by the Alpha architecture. These include code that:
  - Runs in inner access modes or elevated IPL (for example, VAX device drivers)
  - Refers directly to addresses in system space
  - Refers directly to undocumented system services
  - Uses threaded code; for example, code that switches stacks
  - Uses VAX vector instructions
  - Uses privileged VAX instructions
  - Inspects or modifies return addresses or makes other decisions based on a program counter (PC)
  - Depends on exact access-violation behavior due to 512-byte size memory page dependencies
  - Aligns global sections on boundaries other than the native machine page boundary (for example, depends on a 512-byte memory page size)
  - Uses most of the VAX P0 or P1 space or is otherwise sensitive to the space taken up by the translated-image run-time support routines

Although the translated image's run-time performance will be degraded because of the amount of VAX code that TIE will be required to interpret, VEST can probably translate the following kinds of images:

- Images that include self-modifying or created-on-the-fly VAX code, except for the code generated at run time by TIE
- Images with code that inspects the instruction stream, except when TIE interprets such code at run time

For more information on which images can be translated, see *DECmigrate for OpenVMS AXP Systems Translating Images*.

2.4 Coding Practices That Affect Recompilation

Many applications, especially those that use only standard coding practices or are written with portability in mind, will migrate from OpenVMS VAX to OpenVMS Alpha with little or no trouble. However, recompiling an application that depends on VAX specific features that are incompatible with the Alpha architecture will require modifying your source code. Typical incompatibilities include use of the following:

- VAX MACRO assembly language to obtain high performance on a VAX system or to make use of features specific to the VAX architecture
- Privileged code
- Features specific to the VAX architecture

If none of these incompatibilities is present in your application, the rest of this chapter does not apply to you.

2.4.1 VAX MACRO Assembly Language

On Alpha systems, VAX MACRO is not the assembly language, but just another compiled language. However, unlike the high-level language Alpha compilers, the VAX MACRO–32 Compiler for OpenVMS Alpha does not produce highly optimized code in all cases. Digital strongly recommends that you use the VAX MACRO–32 Compiler for OpenVMS Alpha only as a migration aid, not for writing new code.

Many of the reasons for using assembly language on a VAX system are no longer relevant on Alpha systems, for example:

- There is no inherent performance advantage in using assembly language on a RISC processor. RISC compilers, such as those in the Alpha compiler set, can generate optimized code that takes advantage of architecture- and implementation-specific features more easily and efficiently than a programmer can.
- New system services can perform some functions that previously required assembly language.

For more information on migrating MACRO code, see *Porting VAX MACRO Code from OpenVMS VAX to OpenVMS Alpha*.

2.4.2 Privileged Code

VAX code that executes in inner access mode (kernel, executive, or supervisor mode) or that references system space is more likely to use coding practices dependent on the VAX architecture or to refer to VAX data cells that do not exist on OpenVMS Alpha. Such code will not migrate to an Alpha system without change. These programs will require recoding, recompiling, and relinking.

Code in this category includes:

- User-written system services and other privileged shareable images

  For more information, see the *OpenVMS Programming Concepts Manual* and the *OpenVMS Linker Utility Manual*.
- Device drivers and performance monitors not supplied by Digital
- Code that uses special privileges; for example, code that uses $CMEXEC or $CMKRNL system services, or code that uses the $CRMPSC system service with the PFNMAP option

  For more information on memory mapping, see Chapter 6.
- Code that uses internal OpenVMS routines or data, such as:
  - Code that links against the system symbol table, SYS.STB, to access locations in system address space
  - Code that compiles against SYS$LIBRARY:LIB

For assistance in migrating inner-mode code that refers to the OpenVMS executive, contact Multivendor Customer Services.

2.4.3 Features Specific to the VAX Architecture

To achieve its high performance, the Alpha architecture differs significantly from the VAX architecture. Software developers who have become accustomed to writing code that relies on certain aspects of the VAX architecture must be aware of architectural dependencies in their code in order to transport it successfully to an Alpha system.

Common architectural dependencies, along with ways to identify them and actions you can take to eliminate them, are described briefly in the following sections. For a detailed discussion of ways to identify and eliminate these dependencies, see Chapters 5 to 9.

2.5 Identifying Dependencies on the VAX Architecture in Your Application

Even if your application recompiles successfully with a compiler that generates native Alpha code, it may still contain subtle dependencies on the VAX architecture. The OpenVMS Alpha operating system has been designed to provide a high degree of compatibility with OpenVMS VAX; however, the fundamental differences between the VAX and Alpha architectures can create problems for applications that depend on certain VAX architectural features. The following list highlights areas of your application you should examine.

- Check the data declarations contained in your application.

  The high-level language data types you selected to represent data items on a VAX system may not be the best choice on an Alpha system. In particular, consider the following:

  - **Data packing**—Applications on VAX systems typically use the smallest available data type to represent a data item to achieve efficient use of memory resources. For various reasons, using larger data types may be more efficient on OpenVMS Alpha systems. For example, unaligned data can take up to 100 times longer to process than aligned data. For more information, see Chapter 8.
  - **Data-type selection**—The Alpha architecture supports most of the VAX native data types; however, certain VAX data types, such as the H_floating data type, are not supported (see Table 2–1). Check to see if your application depends on the size or bit representation of an underlying native data type.

Table 2–1 Floating-Point Data Type Support

Data Type	On VAX	On Alpha
D53_floating (G_floating) (Default double-precision format)	Not supported.	Supported. Using D53_floating instead of D56_floating drops three bits of precision and yields slightly different results.
D56_floating (Default double-precision format)	Supported.	Not supported. You can obtain full support by translating your code with DECmigrate. Alternatively, you can substitute D53_floating for D56_floating, if your application does not require the extra three bits of precision.

(continued on next page)

Table 2–1 (Cont.) Floating-Point Data Type Support

Data Type	On VAX	On Alpha
F_floating	Supported.	Supported.
G_floating	Supported.	Supported.
H_floating (128-bit floating-point)	Supported.	Not supported. You can obtain full H_floating support with DECmigrate. You can use it to translate the code module that contains H_floating structures, or you can recode your application, using a supported data type.
S_floating (IEEE)	Not supported.	Supported.
T_floating (IEEE)	Not supported.	Supported.
X_floating (128-bit floating-point (Alpha; IEEE-like))	Not supported.	Supported by DEC Fortran Version 6.2 and by DEC C Version 4.0. The X_floating data format is not identical to H_floating, but both cover a similar range of values. For Fortran applications, automatic conversion between X_floating memory format and H_floating on-disk is possible by use of the FOR$CONVERTnnn logical name, the OPTIONS statement, the /CONVERT compiler qualifier, or the CONVERT=keyword on OPEN statements.

- **Shared access to data**—Check any writable data item that is accessed by multiple threads of execution. The VAX architecture includes instructions that can perform certain complex operations, such as incrementing a variable, that appear as a single, noninterruptable operation to other threads of execution. The Alpha architecture is a load-store architecture that does not support atomic memory-to-memory modifications, so different program constructs may be required. Chapter 7 provides more information about this topic.

  In addition, the VAX architecture supports instructions that can manipulate byte- and word-sized data in a single noninterruptable operation. The Alpha architecture supports noninterruptable access only to aligned longword or aligned quadword-sized data. Chapters 7 and 8 describe how this can affect your application.

- **Buffer size**—Your application may determine the size of certain data buffers based on the VAX page size. Different implementations of the Alpha architecture can support 8K, 16K, 32K, or 64K byte pages. Search your application for the text strings 512 and 511 (or the hexadecimal equivalents, 200 and 1FF) to find dependencies on the VAX page size. Chapter 6 describes how to adapt your application to this change in page size.

• Check any condition handlers your application may include.

  While the condition-handling facility on OpenVMS Alpha systems is functionally equivalent to the VAX condition-handling facility, certain aspects of the facility have changed, such as the format of the mechanism array. In addition, the way in which arithmetic exceptions are reported has changed. For more information about this topic, see Chapter 9.

• Check for dependence on the AST parameter list.

While the AST parameter list on OpenVMS Alpha systems has the same format as on VAX systems, only the AST parameter field can be modified. The other fields in the AST parameter list (contents of R0, R1, program counter [PC], and processor status [PS]) are provided for compatibility only and have no subsequent use after the AST procedure exits.

For example, on OpenVMS VAX systems, some user-written AST procedures are designed to change one or more of the values in the other fields in the AST parameter list so that these new values take effect upon completion of the AST procedure. Because ASTs are handled differently on OpenVMS Alpha systems, such changes by the AST procedure to the other fields in the AST parameter list have no effect. Anything an AST procedure writes to the last four parameters on an Alpha computer is lost when the AST procedure exits.

For more information about dependencies on the VAX architecture, see the following sections; for language differences, see Chapter 12 and the user's guides for the particular language you are using. For applications written in VAX MACRO, see *Porting VAX MACRO Code from OpenVMS VAX to OpenVMS Alpha*.

2.5.1 Performance Issues

Two differences between the VAX and Alpha architectures do not keep a VAX application from running on OpenVMS Alpha, but do have a significant performance impact:

- Data alignment
- Choice of data types

2.5.1.1 Data Alignment

Data is **naturally aligned** when its address is an integral multiple of the size of the data in bytes. For example, a longword is naturally aligned at any address that is a multiple of 4, and a quadword is naturally aligned at any address that is a multiple of 8. A structure is naturally aligned when all its members are naturally aligned.

Accessing data that is not naturally aligned in memory incurs a significant performance penalty both on VAX and on Alpha systems. On VAX systems, most languages align data on the next available byte boundary by default, because the VAX architecture provides hardware support that minimizes the performance penalty in referencing unaligned data. On Alpha systems, however, the default is to align each data item naturally, so Alpha, like other typical RISC architectures, does not provide hardware support to minimize the performance degradation from using unaligned data. As a result, references to naturally aligned data on Alpha systems are 10 to 100 times faster than references to unaligned data.

Alpha compilers automatically correct most potential alignment problems and flag others.

Finding the Problem

To find instances of unaligned data, you can use the following methods:

- Use a qualifier provided by most Alpha compilers that allows the compiler to report compile-time references to unaligned data. For example, for DEC C and DEC Fortran programs, compile with the qualifier /WARNING=ALIGNMENT.

- To detect unaligned data at run time, use the OpenVMS Debugger (SET BREAK/UNALIGNED command) or DEC PCA (Performance and Coverage Analyzer).

Eliminating the Problem

To eliminate unaligned data, you will be able to use one or more of the following methods:

- Maximize performance by aligning data items on quadword boundaries, since Alpha systems generally provide only quadword **granularity** (see Section 2.5.2.2).
- Compile with natural alignment, or, when language semantics do not provide for this, move data to be naturally aligned. Where filler is inserted to ensure that data remains aligned, there is a penalty in increased memory size. A useful technique for ensuring naturally aligned data while conserving memory is to declare longer variables first.
- Use high-level-language instructions that force natural alignment within data structures. For example, in DEC C, natural alignment is the default option. To define data structures that must match the VAX C default alignment—such as on-disk data structures—use the construct #PRAGMA NO_MEMBER_ALIGNMENT. With DEC Fortran, local variables are naturally aligned by default. To control alignment of record structures and common blocks, use the /ALIGN qualifier.
- Use compiler qualifiers that generate VAX compatible unaligned data-structure mappings. Use of these qualifiers will result in Alpha programs that are functionally correct but potentially slow.

Note

Software that is converted to natural alignment may be incompatible with other software that is running translated, on a VAX system in the same VMScluster environment, or over a network; for example:

- An existing file format may specify records with unaligned data.
- A translated image may pass unaligned data to, or expect it from, a native image.

In such cases, you will have to adapt all parts of the application to expect the same type of data, either aligned or unaligned.

For more information on data alignment, see Chapter 8 and Section 9.4.2.

2.5.1.2 Data Types

To improve their performance, Alpha processors implement the numeric string and packed decimal string, H_floating, and full-precision D_floating data types by using software, as follows:

- Decimal

  Eighteen-digit decimal data is converted to 64-bit binary integers internally, which provides very fast COBOL performance.
- H_floating

Alpha compilers do not support H_floating data; however, the Translated Image Environment (TIE) provides emulated support for H_floating data in translated images.

- D_floating

  D_floating data is implemented on Alpha platforms in the following ways:

  - Using G_floating hardware (D53). Alpha hardware converts D_floating data (D53) to G_floating for processing. This provides speed and data-type compatibility with existing binary files that contain D_floating data, but loses 3 fraction bits compared to D_floating arithmetic on current VAX systems. D_floating data is thus processed with 15 decimal digits of precision instead of the 16 decimal digits supplied by D56 on a VAX system.
  - Using software emulation (D56) for translated images. This gives exact D56 format VAX results, but is slower than D53 or G_floating.

Eliminating the Problem

To eliminate data type problems, you will be able to use one or more of the following methods:

- Instead of D_floating or H_floating, use G_floating or IEEE T_floating whenever possible because both:
  - Support data in the range 10^{-308} to 10^{308}
  - Have approximately 15 decimal digits of precision
- Instead of decimal data types, use integer data types whenever possible.

For more information on Alpha data types, see Chapter 8.

2.5.2 Protection of Shared Data

Several differences between the VAX and Alpha architectures can affect the integrity of shared data.

2.5.2.1 Modifying Data in Memory

An **atomic operation** is one in which:

- Intermediate or partial results cannot be seen by other processors or devices.
- The operation cannot be interrupted (that is, once started, the operation continues until completion).

On OpenVMS Alpha, any operation that reads, modifies, and stores data in memory will be broken into several instructions, and can be interrupted between any of those instructions. As a result, if your application expects to modify data in shared memory atomically, you must take steps to guarantee the atomicity of the operation.

An application can depend on the atomicity of operations under any of the following conditions:

- An AST routine within the process shares data with the mainline code.
- The process shares data in a **writable global section** with another process that executes on the same CPU (that is, in a uniprocessor system).

- The process shares data in a writable global section with another process that may execute concurrently on another CPU (that is, in a multiprocessor system).

Finding the Problem

To find dependencies on atomicity, reexamine use of shared variables for hidden or explicit assumptions of atomicity.

Eliminating the Problem

To eliminate general problems of instruction atomicity, you will be able to use one or more of the following methods:

- Use language constructs, where available, that guarantee atomicity to protect shared variables: for example, in C, the VOLATILE declaration.
- Use explicit **synchronization** rather than relying on assumptions of atomicity.
- Use OpenVMS locking services (such as $ENQ and $DEQ), Parallel Processing Run-Time Library (PPL$) routines, or LIB$ routines.
- To synchronize with an AST thread, use the $SETAST system service in the mainline code to block the AST and then reenable delivery after the instruction has completed.

For more information on synchronization, see Chapter 7.

2.5.2.2 Reading or Writing Data Smaller Than a Quadword

Granularity refers to the size of the data that can be read or written to memory as an atomic operation, without interfering with data in adjacent memory locations. Machines such as the VAX that provide granularity at the byte level are said to be **byte granular**. Alpha systems are **quadword granular**.[1]

Since OpenVMS Alpha is quadword granular, writes to a shared byte, word, or longword may corrupt other data present in the same quadword as the shared data. This occurs when:

- A program attempts to modify a byte, word, or longword.
- An unaligned field of any size crosses an aligned quadword boundary, which creates a byte, word, or longword that must be written independently.

Note

All of the types of data sharing listed in the discussion of atomicity (Section 2.5.2.1) can create granularity problems in the rest of the quadword containing the intentionally shared data.

In addition, if a process invokes asynchronous system services or asynchronously completing library functions that write a result back to process space, then the data written back can create granularity problems in the quadword that contains it; for example:

- An asynchronous system service that writes to a status block
- An I/O operation that writes to a process buffer

[1] The Alpha architecture also supports longword granularity, but assuming longword granularity is not recommended. Digital compilers assume by default that source code does not depend on granularity finer than quadword, but most Digital languages allow you to specify a smaller granularity by using the /GRANULARITY qualifier.

- An I/O operation in which a direct-memory-access (DMA) controller writes to a process buffer

---

Finding the Problem

To find uses of byte, word, or longword granularity, you can use the following methods:

- Look for intentionally shared data (between an AST and main thread or between processes). Check whether the shared data occupies the same quadword as other data that might be written.
- Look for data written back by asynchronous system services or library calls that complete asynchronously. Check whether that data occupies the same quadword as other data written by another process.
- Look for I/O buffers that contain data written back asynchronously from a device. Check whether the start and end of the buffers occupy the same quadword as data written by another process.

Eliminating the Problem

To eliminate use of granularity at a level smaller than the quadword, you will be able to use one or more of the following methods:

- Put shared items in private quadwords.
- Align I/O buffer heads on quadword boundaries and move any data after the buffer into the next quadword.
- If the problem is not caused by data shared with the system, use a higher-level synchronization mechanism to interlock both intentionally shared data and background data in the same quadword.

Digital compilers assume quadword granularity by default, but to maintain compatibility with your current code, they allow you to specify byte, word, unaligned longword, and unaligned quadword granularity by using the /GRANULARITY qualifier. For more information, see the documentation for the individual compilers.

For more information on read/write granularity, see Chapter 7.

2.5.2.3 Page Size Considerations

Page size governs the amount of virtual memory allocated by memory management routines and system services. It is also the basis on which protection is assigned to code and data in memory.

The OpenVMS VAX operating system allocates memory in multiples of 512 bytes. To improve overall system performance, Alpha systems have much larger page sizes, ranging from 8 KB to 64 KB, depending on the specific hardware platform.

Page size is a factor in the management of system resources, such as working set quotas. In addition, memory protection on VAX systems can vary for each 512-byte region of memory; on Alpha systems, the granularity of memory protection is much larger, depending on the system's page-size implementation.

Note

The change to a larger page size affects only applications that explicitly rely on a 512-byte page size, for example, applications that:

- Use "512" to:
  - Compute memory usage.
  - Calculate page table requirements.
- Change protection on a 512-byte granularity.
- Use the system service Create and Map Section ($CRMPSC) to map a file into a specific location in the process address space (for example, to reuse memory when available memory is limited).

Finding the Problem

To find uses of the VAX page size, identify code that manipulates virtual memory in 512-byte chunks or relies on 512-byte memory protection granularity. Search your code for occurrences of numbers such as the decimal values 511, 512, or 513; the hexadecimal values 1FF, 200, or 201; and so forth.

Eliminating the Problem

To eliminate conflicts between the VAX and Alpha page sizes, you will be able to use one or more of the following methods:

- Change hardcoded page size references to symbolic values (assigned at run time using a call to $GETSYI).
- Reevaluate code that assumes that page size and disk (file) block size are equal. On Alpha systems, this assumption is not correct.
- Do not depend on being able to use memory-management-related system services (for example, $CRMPSC, $MGBLSC) to map a file into a fixed, page-size-dependent range of addresses (global section). Consider instead using the $EXPREG system service.

For more information on page size, see Chapter 6.

2.5.2.4 Order of Read/Write Operations on Multiprocessor Systems

The VAX architecture specifies that if a processor in a multiprocessing system writes multiple data items to memory, these items are written to memory in the order specified. This ordering ensures that the writes become visible to other CPUs in the order in which they were specified by the program and I/O devices.

The guarantee that writes become visible to other CPUs in the same order in which they are specified limits the performance optimization that the system can make. It also makes caches more complex and limits the optimization of cache performance.

To benefit overall system performance, Alpha systems, as well as other RISC systems, can reorder reads and writes to memory. Therefore, writes to memory can become visible to other CPUs in the system in an order different from that in which they were issued.

Note

This section is relevant only to multiprocessor systems. On a uniprocessor system, all memory accesses are completed in the order in which the program requested them.

Finding the Problem

To find instances of reliance on read/write ordering for applications that may execute on multiprocessor systems, identify algorithms that depend upon the order in which data is written: for example, use of flag-passing protocols for synchronization.

Eliminating the Problem

To eliminate problems with the ordering of read and write operations, you will be able to use one or more of the following methods:

- Instead of flag-passing protocols, use system-supplied routines for synchronization, such as those in the Parallel Processing Run-Time Library (PPL$) or the OpenVMS locking system services ($ENQ, $DEQ).
- The Alpha architecture specifies a memory barrier instruction, which causes the hardware to complete all previous memory reads and writes before performing reads and writes following the barrier. Some Alpha languages provide a way of inserting this instruction, but its use will degrade performance.

For more information on synchronization, see Chapter 7.

2.5.3 Immediacy of Arithmetic Exception Reporting

Alpha (and vector VAX) systems treat exceptions differently from scalar VAX systems. Scalar VAX systems use "precise exception reporting;" that is, they guarantee that if an instruction causes an exception, the **program counter (PC)** that is reported is the address of the instruction that caused the exception. Because no subsequent instructions have been issued or have affected the context of the process, a condition handler can remedy the cause of the exception and resume execution of the program at or after the failing instruction.

To achieve the best possible performance in a pipelined environment, vector VAX and Alpha systems use "imprecise exception reporting;" that is, the PC reported by the exception handler is not guaranteed to be that of the instruction that caused the arithmetic exception. Furthermore, subsequent instructions may complete before the exception is reported.

In practice, very few, if any, programs rely on knowing the specific instruction that caused an arithmetic exception. Typically, when an arithmetic exception occurs, a program does one of the following:

- Logs the exception and continues
- Prints an error message and aborts the subroutine or program
- Restarts the entire subroutine and uses a different algorithm that scales the data to prevent overflow or underflow

If a VAX program performs one of these actions upon encountering an arithmetic exception, it will not be affected by being migrated to a RISC system that uses imprecise exception handling.

Note

Imprecise exception reporting applies only to arithmetic exceptions. For other types of exceptions, such as faults and traps, OpenVMS Alpha uses precise exception reporting, and the specific instruction that caused the exception is reported.

Finding the Problem

To find instances of reliance on precise exception reporting, check for the presence of arithmetic exception handlers. Check whether the handler depends on having the exact program counter (PC) or only needs to know what procedure caused the exception.

Eliminating the Problem

To eliminate the use of precise exception handling, avoid writing arithmetic condition handlers that depend upon knowing the exact instruction that caused an arithmetic exception.

For more information on exception handling, see Chapter 9.

For compatibility, most Alpha compilers provide a compiler option that allows a programmer to specify at compile time whether or not precise exception reporting is required (DEC C: /IEEE_MODE; DEC Fortran: /IEEE_MODE or /SYNCHRONOUS_EXCEPTIONS). Precise exception reporting severely impairs the performance of an application. If only certain operations in an application require precise exception reporting, you should use this option to compile only the portions of the application that contain those operations. For more information, see the documentation for the individual compilers.

2.5.4 Explicit Reliance on the VAX Procedure Calling Standard

The OpenVMS calling standard specifies significantly different calling conventions for Alpha programs than for VAX programs. Application programs that depend explicitly on certain details of the VAX procedure calling conventions must be modified to run as native code on an Alpha system. Such dependencies include:

- Code that locates the placement of arguments relative to the argument pointer (AP)

  In many cases, however, the VAX MACRO–32 Compiler for OpenVMS Alpha compensates for this.
- Code that modifies its return address on the stack
- Code that interprets the contents of a call frame

Both VAX and Alpha compilers provide techniques for accessing procedure arguments. If your code uses these standard mechanisms, you can simply recompile it to run correctly on an Alpha system. If your code does not use these mechanisms, you must rewrite it so that it does. For a description of these standard mechanisms, see the *OpenVMS Calling Standard*.

Translated code mimics the behavior of VAX procedure calling. Images that contain the dependencies listed here will execute properly under translation on an Alpha system.

2.5.5 Explicit Reliance on VAX Exception-Handling Mechanisms

The mechanics of exception handling differ between VAX and Alpha systems. Chapter 9 discusses the differences in how arithmetic exceptions are dispatched on VAX and Alpha systems. This section focuses on the mechanisms by which code dynamically establishes a condition handler and by which a condition handler accesses the exception state.

2.5.5.1 Establishing a Dynamic Condition Handler

VAX systems provide a number of ways in which an application can establish a condition handler dynamically at run time. The OpenVMS calling standard facilitates this operation for VAX programs by providing a space at the top of a call frame in which executing code can place the address of a condition handler that is to service exceptions that occur in the context of that frame. However, the OpenVMS calling standard provides no such writable area in the procedure descriptor for Alpha procedures.

For instance, a VAX MACRO program might use the following instruction sequence to move the address of a condition handler into a call frame:

```
MOVAB    HANDLER,(FP)
```

The MACRO–32 Compiler for OpenVMS Alpha parses this statement and generates appropriate Alpha assembly language code that results in the establishment of the condition handler. For more information, see *Porting VAX MACRO Code from OpenVMS VAX to OpenVMS Alpha*.

Note

On VAX systems, the run-time library routine LIB$ESTABLISH and its counterpart LIB$REVERT allow an application to establish and disestablish a condition handler for the current frame. These routines do not exist on an Alpha system; however, compilers may handle these calls properly (such as with FORTRAN intrinsic functions). For more precise information, see Chapter 12 and the documentation for the compilers relevant to your application.

Translated code mimics the VAX mechanism for dynamically establishing a condition handler.

Certain Alpha compilers (and cross-compilers) provide a language-specific mechanism to establish a dynamic condition handler.

For more information on condition handlers, see Chapter 9.

2.5.5.2 Accessing Data in the Signal and Mechanism Arrays

During exception processing, both VAX and Alpha systems push the exception processor status, an exception PC, a signal array, and a mechanism array onto the stack.

Both the signal array and mechanism array have different contents on VAX and Alpha systems; the mechanism array also has different formats on the two platforms. To work properly in either system, a condition handler that accesses data in the signal array or the mechanism array must use the appropriate CHF$ symbols rather than hardcoded offsets. For descriptions of the appropriate CHF$ symbols, see the Bookreader version of the *OpenVMS Programming Concepts Manual*.

Note

The condition handler cannot successfully locate information in the mechanism array by using hardcoded offsets from AP.

2.5.6 Modification of the VAX AST Parameter List

OpenVMS VAX passes five longword arguments to an AST service routine. AST service routines written in VAX MACRO access this information by using offsets from the argument pointer (AP). OpenVMS VAX allows an AST service routine to modify any of these arguments, including the saved registers and the return PC. These modifications can then affect the interrupted program once the AST routine returns.

Although the AST parameter list on Alpha systems also consists of five parameters, the only argument directly intended for the AST procedure is the AST parameter. Although the other arguments are present, they are not used after the AST procedure exits. Because modifying them has no effect on the thread of operation to be resumed at AST exit, a program that relies on such an effect must be changed to use more conventional argument-passing mechanisms to run on an Alpha system.

2.5.7 Explicit Dependency on the Form and Behavior of VAX Instructions

Programs that rely specifically on the execution behavior of VAX MACRO instructions or on binary encoding of VAX instructions must be modified before being recompiled or relinked to run as native code on an Alpha system.

For example, the following coding practices will not work on an Alpha system:

- In VAX MACRO, embedding a block of VAX instructions in a program data area, and modifying a PC to transfer control to this code block
- Examining condition codes or other information in the processor status word (PSW)

For more information on migrating VAX MACRO code, see *Porting VAX MACRO Code from OpenVMS VAX to OpenVMS Alpha*.

2.5.8 Generation of VAX Instructions at Run Time

Creating and executing conventional VAX instructions will not work in native Alpha mode.

VAX instructions created at run time will execute by emulation in a translated image.

For more information on code that generates specific VAX instructions at run time, see *Porting VAX MACRO Code from OpenVMS VAX to OpenVMS Alpha*.

2.6 Identifying Incompatibilities Between VAX and Alpha Systems

To identify architectural incompatibilities in a module of your application, start by doing a test compile of the module using the Alpha compiler. For information on diagnostic compiler switches, see your language processor documentation.

Many modules will compile and run with no errors. If errors occur, you may have to revise the module.

The DEC compilers can produce messages that are very useful for identifying potential porting problems. For example, the MACRO–32 compiler provides the /FLAG qualifier with 10 options. Depending on which options you include, the compiler reports all unaligned stack and memory references, any run-time code generation (such as self-modifying code), branches between routines, or several other conditions.

The DEC Fortran compiler qualifier, /CHECK, produces messages about any of the various options you specify.

Note

Even if a module runs without error in isolation, there may be latent synchronization problems that will turn up only when the module is run together with other parts of the application.

If a module does not run without error after being recompiled and relinked, you can use the following methods to assess what must be revised to make the program run well on an Alpha system:

- Examining the source code

  A code review at this point can avoid many difficulties in the migration process and save a great deal of time and effort in the later stages of migration. To examine your code, use the checklist in Appendix A, as well as the guidelines in Chapter 5. These migration issues are summarized in Section 2.4.

  If a direct code review of your entire application is not practical, an automated search can still be useful: for example, using a combination of DCL SEARCH and an editor to locate and tabulate instances of architectural dependencies.

- Using messages generated by the compiler in your initial test run

  Compilers will give you information on:

  - Differences between VAX and Alpha compilers
  - Data alignment

  Specific compilers may also identify other differences between the VAX and Alpha architectures.

- Analyzing the image using VEST

  Even if you intend to recompile and relink a program, you can use VEST as an analysis tool. It can provide a great deal of useful information about changes that will make your program run most efficiently on an Alpha system. For example, VEST can help identify the following problems:

  - Static unaligned data (data declarations, including unaligned fields in data structures) and unaligned stack operations
  - Floating-point references (H_ and D_floating)
  - Packed decimal references
  - Privileged code
  - Nonstandard coding practice
    - References to OpenVMS data or code other than by using system services
    - Uninitialized variables
  - Certain synchronization issues, such as multiprocess use of interlocked instructions

  VEST cannot identify some problems, including:

  - Unaligned variables (in data structures created dynamically)

  - Most synchronization issues
- Running the image using the PCA (Performance and Coverage Analyzer)

  The PCA can point out the following issues:
  - Run-time alignment faults
  - Which sections of the application are executed most frequently and hence are critical to performance

Once all the images of the application run without errors on an Alpha system, you must combine the rebuilt images to test for problems of synchronization between images. For more information on testing, see Section 4.3.3.

2.7 Deciding Whether to Recompile or Translate

If both methods are possible for a given image, you must balance the projected performance of native and translated versions of the image on an Alpha system against the effort required to translate the image or to convert it to a native Alpha image.

In general, different images that make up an application can be run in different modes: for example, a native Alpha image can call translated shareable images and vice versa. For more information on mixed-architecture applications, see Section 2.7.2.

The two migration paths are compared in Table 2–2.

Table 2–2 Migration Path Comparison

Factor	Recompile/Relink	Translate
Performance	Full Alpha capability	Typically 25-40% of native Alpha potential; equivalent to performance on VAX
Effort required	Varies: easy to difficult	Easy
Schedule constraints	Based on availability of native compilers	None: available immediately
Programs supported		
–Age	Source for VAX/VMS Version 4.0 or earlier accepted	Only VAX/VMS Version 4.0 or later supported
–Limitations	Privileged code supported	Only user-mode code supported
VAX compatibility	High: most code will recompile and relink without difficulty	Complete by emulation
Ongoing support and maintenance	Normal source code maintenance	Source maintenance on VAX; recompile and retranslate for each new version

To determine how to proceed with the migration of your application, answer the following questions:

- Do you build your application entirely from source code, or do you rely on binary images for some functions?

  If you rely on binary images, you will have to translate them.

- Do you have access to the source code for all images that are part of your application?

  If not, you will have to translate those images with missing source code.

- Which images are critical to the performance of your application?

  You will want to recompile those images to take full advantage of the speed of Alpha systems.

  - Use the Performance and Coverage Analyzer to identify critical images.
  - Only images that are produced by native Alpha compilers use Alpha processing capabilities efficiently and achieve optimal performance. A translated VAX image runs at one-third the speed of native Alpha code or slower, depending on the translation options used.

- How much work will be required to convert each image under the two methods?

  - Depending on the complexity of the application, software translation usually requires less effort and time than recompiling and relinking.

    You may choose to translate some part of your application in order to get it running on OpenVMS Alpha while you complete the migration to an all-native version.

  - Code that depends on details of the VAX architecture and the VAX calling standard cannot be recompiled directly. It must either run under translation, or it must be rewritten, recompiled, and relinked.

You can remove architectural dependencies in several ways:

- Replace an architecture-dependent code sequence with high-level language lexical elements that support the same operation in a platform-independent manner.
- Use a call to an OpenVMS system service to perform the task in a way that is appropriate for the processor architecture.
- Use a high-level language compiler switch to help guarantee correct program behavior with minimal changes to the source code.

Table 2–3 summarizes how the architectural dependencies of a given program can affect which method you use to migrate the program to an Alpha system. For more detailed information, see the following chapters.

Table 2–3 Choice of Migration Method: Dealing with Architectural Dependencies

Recompiled, Relinked VAX Source	Translated VAX Image
Data alignment[1]	
By default, most compilers align data naturally. For information on qualifiers to retain VAX alignment, see Chapter 12.	Unaligned data supported, but the qualifier /OPTIMIZE=ALIGNMENT can improve overall execution speed by assuming that data is longword aligned.
Data types	
Replace H_floating with X_floating. For D_floating, if the 15 decimal digits of precision provided by the D53 format are sufficient, replace D_floating with G_floating. If the application requires 16-bit decimal precision (D56 format), translate it. COBOL packed decimal is automatically converted to binary format for operations. For more information on data types, see Chapter 8.	To retain 16-bit decimal precision for D_floating, use the /FLOAT=D56_FLOAT qualifier. Performance using this qualifier will be slower than when using the default, /FLOAT=D53_FLOAT.
Atomicity of read-modify-write operations	
Support depends on options provided by the individual compiler. (For more information, see Chapter 7.)	Use the /PRESERVE=INSTRUCTION_ATOMICITY qualifier. Execution speed may drop by a factor of 2.
Atomicity and granularity of byte and word write operations	
Supported using compiler options with appropriate source code changes. (For more information, see Chapter 7.)	Use the /PRESERVE=MEMORY_ATOMICITY qualifier. Execution speed may drop by a factor of 2.
Page size	
The OpenVMS Linker produces large, Alpha style pages by default.	Most 512-byte page images are supported. However, because of the permissive protection assigned by VEST, images that rely on restrictive protection to generate access violations will not execute properly on an Alpha system when translated.

[1]Unaligned data is primarily a performance issue. Whereas references to unaligned data were only somewhat detrimental to VAX performance, loading unaligned data from memory and storing unaligned data to memory in an Alpha system can be up to 100 times slower than the corresponding aligned operations.

(continued on next page)

Table 2–3 (Cont.) Choice of Migration Method: Dealing with Architectural Dependencies

Recompiled, Relinked VAX Source	Translated VAX Image
Read/write ordering	
Supported by adding appropriate synchronization instructions (MB) to source code, but with a performance penalty. (For more information, see Chapter 7.)	Use the /PRESERVE=READ_WRITE_ ORDERING qualifier.
Immediacy of exception reporting	
Partly supported using compiler options. (For more information, see Chapter 9.)	Use the /PRESERVE=FLOAT_EXCEPTIONS or the /PRESERVE=INTEGER_EXCEPTIONS qualifier. Execution speed may drop by a factor of 2.
Explicit reliance on details of the VAX architecture and calling standard[2]	
Unsupported; dependencies must be removed.	Supported.

[2]Dependencies on details of the VAX architecture and calling standard include explicit reliance on the VAX calling standard, VAX exception handling, the VAX AST parameter list, the format and behavior of VAX instructions, and the generation of VAX instructions at run time.

2.7.1 Translating Your Application

If you are unable to recompile your application, or if it uses features specific to the VAX architecture, you may decide to translate the application. You can choose to translate only some parts of the application, or you can translate parts of it temporarily as a means of staging the overall migration.

Many of the differences that affect recompilation discussed in Section 2.4 can also affect the performance of a translated VAX image. You can use the following methods to increase the compatibility of images that are dependent on the VAX architecture. (For more information, see *DECmigrate for OpenVMS AXP Systems Translating Images*.)

- Data alignment

  Supply the VEST translate-time qualifier /OPTIMIZE=NOALIGNMENT to make VEST generate extra inline Alpha code that avoids generating exceptions for references to unaligned data. With this qualifier, VEST produces Alpha code that executes about 10 times slower than code that uses only aligned data references. (If you use the default option /OPTIMIZE=ALIGNMENT, unaligned data causes an exception, which takes about 100 times longer to execute than with aligned data.)

- Instruction atomicity

  When you invoke the translator, supply the translate-time qualifier /PRESERVE=INSTRUCTION_ATOMICITY to make VEST generate an Alpha instruction sequence that is AST atomic for a specified set of VAX instructions. Although an AST can be delivered in the middle of an Alpha instruction sequence that performs such an atomic operation, the instruction sequence will be restarted at the beginning when the AST routine completes.

Execution speed for a particular code sequence may drop by a factor of 2 if the /PRESERVE=INSTRUCTION_ATOMICITY qualifier is specified. (For a list of VAX instructions for which the translator generates AST-atomic code, as well as additional information about the software translator, see *DECmigrate for OpenVMS AXP Systems Translating Images*.)

- Read/write granularity

  VEST ensures the atomicity of byte or word writes when you use the translate-time qualifier /PRESERVE=MEMORY_ATOMICITY. This qualifier allows a mainline routine and an AST routine to update adjacent bytes within a longword or quadword concurrently without interfering with each other. The /PRESERVE=MEMORY_ATOMICITY qualifier guarantees atomic access of longwords that are not naturally aligned and of data that crosses quadword boundaries. Execution speed may drop by a factor of 2 when these qualifiers are specified.

- Page size and permissive protection

  To enable VAX images to run on an Alpha system, VEST, together with the image activator, maps the VAX image sections into large pages. With an Alpha processor that supports 8 KB pages, up to 16 VAX pages can fit in a single page. However, because this big page is described by only a single page-table entry, only one protection and a single backing store can be assigned to the page. Consequently, VEST assigns the Alpha page the most permissive protection associated with any of the Alpha image sections that it maps. Thus, VAX images that rely on restrictive protection to generate access violations will not execute properly on an Alpha system when translated.

  One possible alternative is to relink the program on a VAX using the default linker qualifier /BPAGE to align the pages on 64KB boundaries.

- Precise arithmetic exceptions

  VEST allows you to set precise exception reporting for certain exception types at translate time by using the /PRESERVE=FLOAT_EXCEPTIONS qualifier or the /PRESERVE=INTEGER_EXCEPTIONS qualifier. If you specify either of these qualifiers, execution speed for certain code segments may drop by a factor of 2.

- Generating VAX instructions at run time

  VAX instructions created at run time will execute by emulation under translation. However, emulated instructions are significantly slower than translated instructions, which can be important if the code is generated at run time to speed up the performance of critical sections of your application.

Table 2–3 includes a summary of ways you can allow for various architectural dependencies in a translated image.

2.7.2 Combining Native and Translated Images

In general, you can combine native Alpha images with translated images on an Alpha system. For example, a native Alpha image can call translated shareable images and vice versa.

In order to run together, native and translated images must be able to make calls between the VAX and Alpha calling standards. No special action is required if the native and translated images meet the following conditions:

- Routine interface semantics and data alignment conventions for the native Alpha image are identical to those on a VAX image.
- All the entry points are CALL*x*; that is, there are no external JSB entry points. This is probably true of any code written in a high-level language.
- The inbound and outbound calls in the native image are not written in Ada.

When a source procedure that uses one calling standard calls a destination procedure that uses a different calling standard, it does so indirectly through a **jacket routine**. The jacket routine interprets the procedure's call frame and argument list and builds the equivalent destination call frame and argument list, then transfers control to the destination procedure. When the destination procedure returns, it does so through the jacket routine. The jacket routine propagates the destination routine's returned register values into the source routine's registers and returns control to the source procedure.

The OpenVMS Alpha operating system creates jacket routines automatically for most calls. To make use of automatic jacketing, use the compiler qualifier /TIE and the linker option /NONATIVE_ONLY to create the native Alpha parts of your application.

In certain cases, the application program must use a specially written jacket routine. For example, you may have to write jacket routines for nonstandard calls to libraries such as the following:

- A VAX shareable library that includes external JSB entry points
- A library that includes read/write data in the transfer vector
- A library that contains VAX specific functions
- A library that uses resources that would need to be shared between a native and a translated version of the library
- A native Alpha library that does not provide or export all the symbols that the VAX image did

  (The term exported means that a routine is included in the Global Symbol Table (GST) for the image.)

For information on how to create a jacket image for one of these situations, see *DECmigrate for OpenVMS AXP Systems Translating Images*.

Translated shareable images (such as run-time libraries for languages without native Alpha compilers) that are shipped with the OpenVMS Alpha operating system are accompanied by jacket routines that allow them to be called by native Alpha images.

3
Sample Migration Plan

The following migration plan is for a fictitious application, but is based on actual migration plans written for customer applications.

Migration Plan for Omega-1

Omega Corporation

3.1 Executive Summary

Omega-1 is an enterprise-wide information system for accessing, analyzing, managing, and presenting data.

Omega-1 has more than 4 million lines of source code. Most of the source code, written in the C programming language, is common to a variety of platforms and is considered highly portable.

However, Omega-1 has a set of routines, unique to each platform, that is implemented in VAX C and VAX MACRO for the VAX platform (about 350,000 lines of code). These routines present a number of VAX architectural dependency issues, described in Section 3.2.4, that require resolution for successful migration. Resolution of these issues will involve significant work including design changes, but none of these appear to jeopardize shipping Omega-1 to customers in December 1992.

Omega-1 supports connections to a number of Digital and third-party products. Although some of these products will not be available on the Alpha platform when OpenVMS Alpha is first shipped, Omega Corporation, the Omega-1 vendor, has committed to migrating the base Omega-1 by that date.

Digital Services will provide support services to Omega Corporation throughout the life of the migration project as detailed in this plan. In summary, the support plan for Omega Corporation includes:

- On-site hardware and software tools for Alpha development at Omega Corporation
- Engineering assistance for quality assurance testing
- Access to Alpha systems at an Alpha Migration Center
- Telephone access to a Digital engineer who will provide Alpha technical information, support the cross-development tools, and act as a liaison for resolving any problems with Digital software products reported by Omega Corporation

- A three-day technical seminar for Omega Corporation developers at their site

One additional problem is providing the hardware for Omega to carry out an adequate field test of their products prior to the ship date. Normally, the Omega developers conduct field tests for four months before revenue shipment at 30 to 40 of their customer sites. It is unclear whether the number of required Alpha units will be available for a field test of this size.

3.2 Technical Analysis

The technical analysis was performed by Omega Corporation in conjunction with Digital Services.

3.2.1 Application Characteristics

Omega-1 runs on most VAX platforms and platforms of other vendors. It consists of three layers that may or may not take advantage of specific aspects of the VAX environment. However, there are no direct dependencies on particular hardware configurations or devices.

Most of the functionality is provided in the applications layer, which contains the user interface, basic data management tools, and the Omega fourth generation language (4GL). The Omega-1 base product does not depend on any Digital or third-party layered products.

Additional products in Omega-1 are layered on top of the base product and provide expanded data management or communications functionality. These options depend on Digital or third-party layered products and will be available when the underlying products are released. The layered product dependencies are listed in Table 3–3.

3.2.2 Software Architecture

Omega-1 is built in layers as shown in Figure 3–1. This layering creates a high degree of portability for the software, because only about 10% of the system is specific to a particular implementation, and all of this code is contained within one set of modules, the host layer.

The applications and core layers are expected to run on the Alpha platform simply by recompiling and relinking all of the source files. The only prerequisite is successful migration of the Omega software development tools, which are also considered quite portable and depend only on the C compiler and run-time libraries for OpenVMS Alpha.

The host layer will require a number of changes such as a rewrite of some modules that contain some VAX hardware dependencies.

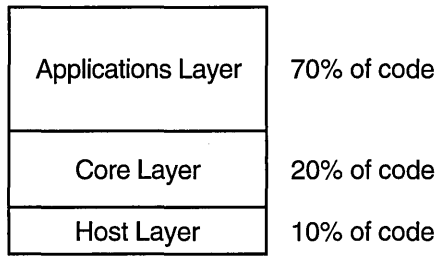
- Applications layer

  This layer comprises the bulk of the system and is considered portable because it is implemented similarly on many platforms.

- Core layer

  This layer creates an Omega-designed set of services that conforms to the special needs of Omega-1 on each platform. Essentially, anything that is part of the Omega-1 "virtual operating system" but can be written portably resides in this layer. Components include high-level I/O, high-level memory

Figure 3–1 Layer Structure of Omega-1



ZK–5174A–GE

management, character-based window systems, and the 4GL compiler with its execution environment. This layer is portable.

- Host layer

  This layer provides an interface to specific operating system elements and may be dependent on aspects of the hardware architecture. Components include:

  - I/O operation
  - Image loading and unloading
  - Memory management
  - Lightweight thread management
  - Terminal services
  - Windowing interfaces

  The host layer is different for each platform implementation. The VAX implementation is written in VAX C and VAX MACRO. This layer embodies most of the issues on which the migration project will focus.

3.2.3 Results of Image Analysis

Although Omega-1 will be recompiled and relinked, the VAX Environment Software Translator (VEST) was used to analyze 26 images of the host layer. A large number of these images had instructions that relied on the D_floating data type, which is the default for VAX C. If Omega engineers decide to move from D_floating to another floating-point format, they must be aware of the issues concerning compatibility of data files across mixed VAX and Alpha VMScluster environments.

Table 3–1 lists the error messages generated by the images during image analysis.

Table 3–1 Image Analysis Results

Image Name	% Code Found by VEST	Major Findings
OMEGADEV60	-Fatal errors- no code found	VEST-F-PROTISD VEST-F-ISDALIGN VEST-F-ISDMIXED VEST-W-STACKMATCH Packed decimal instructions
OMECRTL	92%	VEST-W-STACKMATCH VEST-W-STKUNAL Packed decimal instructions
PSCN	64%	VEST-W-STKUNAL VEST-W-STACKMATCH
PVSN	65%	VEST-W-STKUNAL VEST-W-STACKMATCH

The following list describes the major findings of the Omega image analysis:

- PROTISD—user-written system service vector that indicates that an image has one or more user-written system services. This problem will be handled automatically when compiling the code using a native Alpha compiler.
- ISDALIGN—the image section is not aligned on a 64 KB boundary. Before performing another VEST analysis, it will be necessary to relink the image with 64KB pages. The linker will handle this problem during the migration process.
- ISDMIXED—incompatible VAX image sections were mapped to the same 64KB block. The linker will handle this problem during the migration process.
- STKUNAL—a warning indicates that a block of code changes the stack from longword aligned to unaligned, which causes performance degradation. Omega engineers will review the logs from the VEST analysis.
- STACKMATCH—the stack may be unbalanced at a certain point. Omega engineers will review the logs from the VEST analysis.
- Packed decimal instructions—supported only with software and not with hardware, which may hinder performance of the application. Omega engineers will review the packed decimal code.

3.2.4 Results of Source Analysis

As stated previously, the migration of the applications and core layers is fairly straightforward; however, the host layer of Omega-1 contains many VAX dependencies. Discussions with Omega engineers uncovered the architectural dependencies described in the following list:

- Data alignment

  The Omega-1 software contains unaligned public data structures. To maintain source code portability, Omega engineers will compile this code with the /NOMEMBER_ALIGN qualifier of the DEC C compiler.

- Data types

Omega-1 extensively uses floating-point calculations and data files. The VAX version uses the D56 format for all operations, which is not implemented in the native Alpha instruction set. For customers who may eventually operate with mixed VAX and Alpha clusters, it is best to maintain the D_floating format. Omega is not concerned about the slight loss of floating-point precision entailed in using the 53-bit (versus 56-bit) version of D_floating available on Alpha.

However, most of the other Omega-1 implementations use the IEEE floating-point formats, which are fully supported by the Alpha instruction set. Reimplementation to the IEEE format involves simply recompiling with a different qualifier, but VAX customers would then need to convert all floating-point data in their files to the new format as part of their migration process.

- Read/write/modify atomicity

  A few AST routines need to be examined to determine whether any operations on shared variables need to be protected by explicit synchronization methods. No major problems are expected in this area.

- Granularity of byte and word operations

  The Omega-1 software has a latch that protects data that is not aligned on natural quadword boundaries. Digital engineers discussed this problem with Omega engineers and reviewed a code sample for the solution. They determined that the compilers can handle this type of access using shared data declarations and compiler directives.

- VAX page size

  The host layer includes a few routines that handle memory management on behalf of the applications. Although the existing algorithms do not actually require that the page size be 512 bytes, nevertheless, they are hardcoded as 512. Correct functionality will be guaranteed by modifying these modules to query for system page size at system startup and then using the system page size for calculations involving memory management operations.

- VAX procedure calling standard

  Omega-1 "chases" the call frame stack to determine call history when a user interrupt occurs. Omega-1 modifies the return address in one of the preceding call frames to redirect flow-of-control, or accept the interrupt, when in a noncritical region of the code. Much of this is similar to what SYS$UNWIND does, except that Omega-1 does it with an AST instead of an exception handler.

  Omega-1 includes a number of functions that depend on the VAX calling standard, including Omega's own implementation of setjmp( ) and longjmp( ). These functions are written in VAX MACRO and are isolated in the operating-system code. Omega will rewrite these functions to remove dependencies on the VAX calling standard.

- Exception handling

  Omega-1 fixes illegal or faulted floating-point operations with a statistical "missing" value. It is a concern whether the current design can correctly decode the actual faulting instruction on the Alpha architecture, where delivery of exception traps may be delayed.

- VAX instruction set and code generation

The host layer includes a code generator that writes platform-native instructions into memory and executes them as part of its 4GL language. This code generator produces, among other things, "scatter/gather" code to handle database I/O operations. A portable interpreter that is "plug compatible" with the code generator can be used for the early stages of the migration project. However, a new version of the generator that produces Alpha instructions will eventually have to be implemented.

3.3 Milestones and Deliverables

The ship date goal for the Omega-1 base product is December 1992. Table 3–2 lists the major milestones and deliverables for the base product project. For a discussion of each of the deliverables, see Section 3.4.

Table 3–2 Milestones and Deliverables

Milestone	Responsible	Digital Role	Completion Date
Omega-1 line-mode prompt	Omega/Digital	Consulting	November 1991
New cross-image bridge	Omega/Digital	Consulting	December 1991
Floating-point decision	Omega	—	December 1991
Omega-1 exception handler	Omega/Digital	Consulting	January 1992
Begin code generator	Omega/Digital	Consulting	January 1992
Build applications layer	Omega/Digital	Consulting	January 1992
Test code generator	Omega/Digital	Run test scripts	March 1992
Test applications	Omega/Digital	Run test scripts	May 1992
Implement and test Motif user interface	Omega/Digital	Run test scripts	July 1992
Begin Omega QA and field test	Omega	On-site support	December 1992
Ship date	Omega	—	December 1992

3.4 Technical Approach

The following sections describe in detail the approach to be taken to reach each milestone of this migration project.

3.4.1 Line Mode Prompt

The first milestone is bringing Omega-1 to a line-mode prompt and entering a meaningful Omega-1 program name or command sequence for execution. Reaching this goal will demonstrate basic functionality of the development tools and run-time libraries. At this point, the host layer will be functional except for the following:

- The Omega-1 interpreter will be used instead of code generation for the 4GL functionality.
- The image bridge will be a temporary implementation.
- Exception handling will be incomplete.

Furthermore, the core layer will be functional (without the windowing user interface), and at least one Omega-1 application will be tried. This work is being done by the Omega VMS Host Group with support from Digital.

3.4.2 Image Bridge

Omega-1 has a central bridge routine that dispatches all jumps across images. This allows Omega-1 to call routines in unloaded images, which will then be loaded dynamically. The bridge also allows images to be unloaded by Omega-1.

The image activation routines and the format of Alpha object files have already established that the bridge for OpenVMS Alpha can be implemented similarly to its implementation on OpenVMS VAX.

The work will be done by the Omega VMS Host Group with support from Digital, and the required changes can be completed by December 1991.

3.4.3 Floating-Point Format Decision

Omega-1 and many customers that use it rely on the D56 floating-point data type. Although it is possible to replace D56 with IEEE T_floating for increased speed, this will require that all users convert their data from one format to the other.

This is an Omega business decision, which will be made by the end of calendar year 1991.

3.4.4 Full Omega-1 Exception Handling

The next major issue to be resolved is how to perform Omega-1 exception handling on OpenVMS Alpha. General exception handling, such as a run-time access violation upon opening a file, is trapped by the Omega-1 exception handler, which may then proceed with a "stack chase" on the call frames. Omega developers will make the necessary changes to account for the new calling standard on OpenVMS Alpha, and will use the "setjmp" and "longjmp" features of DEC C.

Floating-point exception handling will also require design changes and reimplementation. There are a number of options for getting correct functionality, but the best answer for performance considerations is yet to be determined. Still, the Omega developers expect to solve this by the scheduled date.

3.4.5 Begin Code Generator Implementation

The final component required for a full implementation of the host layer is the code generator. To implement the code generator, the Omega developers need a complete definition of the native Alpha instruction set. The code generator effort will be fully handled by the Omega VMS Host Group. Digital will provide support by telephone, as needed.

3.4.6 Build Applications

Once the host layer has achieved full functionality, the applications can be built during January 1992. These applications are expected to recompile and run, since they are all written in C to very strict portability standards. Digital will provide telephone support for tools and compiler issues to the Omega developers.

3.4.7 Test Code Generator

Digital will provide assistance with debugging and functional testing for the Omega-1 code generator on an Alpha system during the month of March 1992. An Omega developer will establish a test bed environment on the Alpha system in an Alpha Migration Center, train a Digital engineer how to run the tests, and supervise the initial set of tests. After that, Omega will send test scripts and data sets by mail for the Digital engineer to run.

3.4.8 Test Complete Application

Omega maintains a number of developmental regression test streams, which must be run on an Alpha system to verify successful porting. If Omega does not have an Alpha system by the time they are ready for this testing, Digital will run the regression tests against the Omega-1 base system applications on one of Digital's systems.

To do this, an Omega developer will establish a test bed environment on an Alpha system in the Digital laboratory, train a Digital engineer how to run the regression tests, and supervise the initial set of tests. After that, Omega will send test scripts and data sets by mail for the Digital engineer to run. Digital will then report results back to Omega. This effort will begin in April 1992 and will be finished by the end of May.

3.4.9 DECwindows Motif User Interface

Omega needs to have the Motif developers tool kit prior to field test, so that the developers can test their Motif user interface.

3.4.10 Omega Quality Assurance and Field Test

Omega maintains an extensive set of test streams used to validate their final product. They routinely run these suites immediately prior to starting their field tests. These tests must be run on a full Alpha system implementation.

At this point, some testing and optimization may be required to fix specific performance problems that become apparent only on an Alpha system. Digital will provide on-site engineering support for these efforts.

3.5 Dependencies and Risks

The chief risk to the successful delivery of Omega-1 is related to Omega's quality assurance and field test processes. The developers normally conduct their field test with 30 to 40 of their customers, and it takes about four months to complete. Omega and its Digital account team need to determine how Omega can execute a testing program that meets its minimum requirements and that can be completed before December 1992.

The following list shows the software dependencies for the Omega-1 base product:

- DEC C for OpenVMS Alpha compiler
- VAX MACRO–32 Compiler for OpenVMS Alpha
- MACRO–64 Assembler for OpenVMS Alpha
- OpenVMS Debugger
- DEC C Run-Time Library
- OpenVMS Run-Time Library (LIB$)
- Screen Management Library (SMG$)
- DECTPU

Omega-1 applications also offer access to Digital or third-party data management or networking options. For example:

- Omega/Graph can use CDA tools to generate graphic images in DDIF format.
- Omega/Access can access Rdb/VMS or ORACLE databases.

- Omega/Share and Omega/Connection can access remote data with TCP/IP using the ULTRIX Connection (UCX).

Table 3–3 lists all of the layered product options available to the users of Omega-1 along with the expected delivery dates.

Table 3–3 Omega Optional Product Dependencies

Item	Field Test	Shipping
Digital Products		
DECwindows Motif user interface (tool kit)	TBS	TBS
CDA	TBS	TBS
ALL–IN–1	TBS	TBS
Rdb/VMS	TBS	TBS
CDD/Repository	TBS	TBS
CDD/Plus	TBS	TBS
DECnet (Phase IV)	TBS	TBS
PATHWORKS	TBS	TBS
SPM	TBS	TBS
Third-Party Products		
ORACLE	TBS	TBS
INGRES	TBS	TBS

3.6 Resource Requirements

Digital resources used to support the plan outlined in Section 3.3 are summarized in Table 3–4 and are described in the following sections.

Table 3–4 Summary of Digital Support

Resource	Time Frame	Activity	Level of Effort
On-site training	Dec 91	Training	1 engineer for 3 days
Telephone support	Dec 91–Aug 92	General support	1 engineer for 8 hours/week
Engineering assistance	Mar 92	Test code generator	1 full-time engineer for 2 weeks, half time for 4 weeks
Alpha hardware	Mar 92	Test code generator	5 days/week for two weeks, 2 days/week for 4 weeks

(continued on next page)

Table 3–4 (Cont.) Summary of Digital Support

Resource	Time Frame	Activity	Level of Effort
Engineering assistance	Apr–May 92	Application testing	1 engineer, half time for 8 weeks
Alpha hardware	Apr–May 92	Application testing	2 days/week for 8 weeks
On-site support	Jan–Aug 92	Omega QA	1 week per month

3.6.1 Hardware

Omega requires that a system be loaned to its site to complete the migration tasks without affecting normal development activities.

The Omega-1 base product spans several RA90 drives on its development system, which is a VAX 6000 Model 550. Disk space may become a problem for building the full application set.

3.6.2 On-Site Training

Digital will assume an active role in support of Omega migration beginning in December 1991 with a three-day Alpha technical seminar for the application and core-layer developers at Omega.

3.6.3 Telephone Support

During the first two-thirds of 1992, the Omega developers will continue their efforts to migrate the host layer at their site using the cross tools on a Digital-supplied platform. During this period, and throughout the entire migration effort, Omega will receive telephone support from a software engineer at Digital. The assigned engineer will spend approximately eight hours per week working with Omega issues, following up on bug reports, and supporting the cross tools.

3.6.4 Testing Assistance

Digital will support several phases of testing the Omega-1 application.

3.6.4.1 Testing the Code Generator

One full-time Digital engineer is required for the first two weeks of March to test the code generator. This period includes training by an Omega developer and running the initial tests. During the following four weeks, the assigned engineer will spend 50 percent of working time performing follow-up tests and reporting the results to Omega.

The code generator testing will require nearly full-time use of an Alpha system during the first two weeks, followed by two additional days of Alpha hardware time per week during the following four weeks.

3.6.4.2 Testing Applications

Application testing will be done at Digital during the months of April and May 1992, and will require 40 percent of an engineer's time for running regression tests and reporting results to Omega. This effort will require two days on Alpha hardware per week during the eight-week test period.

3.6.4.3 Omega Quality Assurance

A Digital engineer will be available for an estimated 15 days during the months of January to August 1992 to provide on-site support at Omega.

Part II

Migrating the Application

4
Migrating Your Application

Actually migrating your application to an Alpha system involves several steps:

1. Setting up the migration environment
2. Testing the application on a VAX system to establish baselines for evaluating the migration
3. Converting the application to run on an Alpha system
4. Debugging and testing the migrated application
5. Integrating the migrated application into a software system

4.1 Setting Up the Migration Environment

The native Alpha environment is a complete development environment equivalent to that on VAX systems.

At present, you will have to complete the debugging and testing of your migrated application on Alpha hardware.

An important element of the Alpha migration environment is support from Digital, which can provide help in modifying, debugging, and testing your application.

4.1.1 Hardware

There are several issues to consider when planning what hardware you will need for your migration. To begin, consider what resources are required in your normal VAX development environment:

- CPUs
- Disks
- Memory

To estimate the resources needed for an Alpha migration environment, consider the following issues:

- Greater image size on Alpha systems

  Compare VAX and Alpha compiled and translated images.
- Greater page size and physical memory size on Alpha systems
- CPU requirements

Using VEST tends to take a lot of CPU time. (It is difficult to predict how much; it depends more on application complexity than size.) VEST also needs a great deal of disk space for log files, for an Alpha image if you request one, for flowgraphs, and so on. The new image includes both the original VAX instructions and the new Alpha instructions, so it is always larger than the VAX image.

A suggested configuration consists of:

- 6 VUP multiprocessing system with 256 MB of memory
- 1 GB system disk
- 1 GB disk per application

In a multiprocessing system, each processor should be able to support the image analysis of a separate application.

If computer resources are scarce, Digital suggests that you do one or more of the following:

- Run compilers or VEST as a batch job at off-peak hours.
- Lease additional equipment for the migration effort.

4.1.2 Software

To create an efficient migration environment, check the following elements:

- Migration tools

  You need a compatible set of migration tools, including the following:

  - Compilers
  - Translation tools
    - VEST and VEST/DEPENDENCY
    - TIE
  - RTLs
  - System libraries
  - Include files for C programs
- Logical names

  Logical names must be consistently defined to point to VAX and Alpha versions of tools and files. For more information, see Section 4.4.
- Compile and link procedures

  These procedures must be adjusted for new tools and the new environment.
- Tools for maintaining sources and building images
  - CMS
  - MMS

Native Alpha Development

All of the standard development tools you have on VAX are also available as native tools on Alpha systems.

Translation

The software translator VEST runs on both VAX and Alpha systems. The Translated Image Environment (TIE), which is required to run a translated image, is part of OpenVMS Alpha, so final testing of a translated image must either be done on an Alpha system or at an Alpha Migration Center.

4.2 Converting Your Application

If you have thoroughly analyzed your code and planned the migration process, this final stage should be fairly straightforward. You may be able to recompile or translate many programs with no change. Programs that do not recompile or translate directly will frequently need only straightforward changes to get them up on an Alpha system.

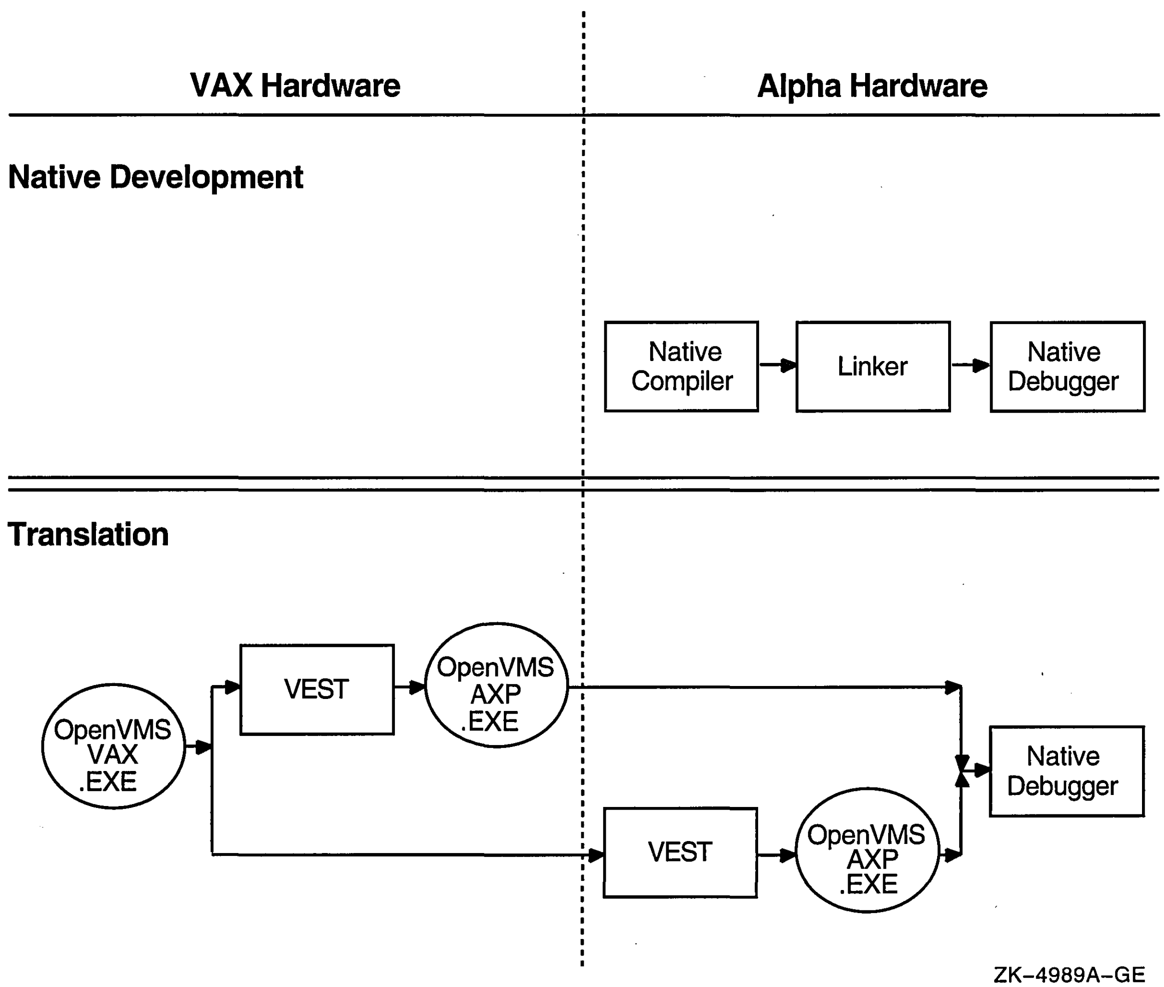
For more detailed information on the actual conversion of your code, see the following OpenVMS Alpha migration documentation:

- *DECmigrate for OpenVMS AXP Systems Translating Images*
- *Porting VAX MACRO Code from OpenVMS VAX to OpenVMS Alpha*

For descriptions of these books, see the Preface of this manual.

The two migration environments and the principal tools used in each are shown in Figure 4–1.

Figure 4–1 Migration Environments and Tools



4.2.1 Recompiling and Relinking

In general, migrating your application involves repeated cycles of revising, compiling, linking, and debugging your code. During the process, you will resolve all syntax and logic errors noted by the development tools. Syntax errors are usually simple to fix; logic errors typically require significant modifications to your code.

Your compile and link commands will require some changes, such as new compiler and linker switches. For example, to allow portability among different Alpha platforms, the linker default page size for Alpha systems is 64 KB, which allows any OpenVMS Alpha image to run on any Alpha processor, regardless of the system page size on that processor. Also, Alpha shareable images declare their universal entry points and symbols by means of a symbol vector declaration in a linker options file, not by means of the VAX transfer vector mechanism.

A number of native compilers and other tools are available for software development and migration on an Alpha platform.

4.2.1.1 Native Alpha Compilers

Recompiling and relinking an existing VAX source produces a native Alpha image that executes within the Alpha environment with all the performance advantages of a RISC architecture. For Alpha code, Digital is using a series of highly optimizing compilers. These compilers have a common optimizing code generator. However, they use a different front end for each language, each of which is compatible with a current VAX compiler.

For OpenVMS Alpha Version 7.0, native Alpha compilers are available for the following languages:

- Ada
- BASIC
- C
- C++
- COBOL
- FORTRAN
- Digital Fortran (Alpha systems only)
- Pascal
- PL/I
- MACRO-32 (cross compiler)

Later releases of OpenVMS Alpha will provide native compilers for other languages, including LISP.

VAX user-mode programs that are written in any other language can be run on an Alpha system by translating them with VEST. Compilers for other languages may be available through third-party vendors.

In general, the Alpha compilers provide command-line qualifiers and language semantics to allow code with dependencies on the VAX architecture to run on an Alpha system with little modification. For a list of such dependencies, see Table 2–3.

Some compilers on OpenVMS Alpha systems support new features not supported by their counterparts on OpenVMS VAX systems. To provide compatibility, some compilers support compatibility modes. For example, the DEC C compiler for OpenVMS Alpha systems supports a VAX C compatibility mode that is invoked by specifying the /STANDARD=VAXC qualifier.

In some cases, the compatibility is limited. For example, VAX C implements built-in functions that allow access to special VAX hardware features. Since the hardware architecture of VAX computers differs from Alpha computers, these built-ins are not available in DEC C for OpenVMS Alpha systems even when the /STANDARD=VAXC qualifier is used.

The compilers can also compensate for some architectural dependencies that may exist in your code. For example, the MACRO–32 compiler provides the /PRESERVE qualifier that can preserve granularity or atomicity or both.

The DEC C compiler provides a header file that defines macros for each data type. These macros map a generic data-type name, such as int64, to the machine-specific data type, such as *-64*. For example, if you must have a data type that is 64 bits long, use the int64 macro.

Review the documentation for your compiler to become familiar with all its features that support portability.

Chapter 12 describes in greater detail the process of using Alpha compilers to migrate OpenVMS VAX programs to an OpenVMS Alpha system.

4.2.1.2 VAX MACRO–32 Compiler for OpenVMS Alpha

The VAX MACRO–32 Compiler for OpenVMS Alpha is used to convert existing VAX MACRO code into machine code that runs on OpenVMS Alpha systems. It is included with OpenVMS Alpha for that purpose.

While some VAX MACRO code can be compiled without any changes, most code modules will require the addition of entry point directives. Many code modules will require other changes as well.

Note

The MACRO–32 compiler will attempt to call LIB$ESTABLISH if it is contained in the source code.

If MACRO–32 programs establish dynamic handlers by storing a routine address at 0(FP), they will work correctly when compiled on an OpenVMS Alpha system. However, you cannot set the condition handler address from within a JSB (Jump to Subroutine) routine, only from within a CALL_ENTRY routine.

The compiler generates code that is optimized for OpenVMS Alpha systems, but many features of the VAX MACRO language that provide the programmer with a high level of control make it more difficult to generate optimal code for OpenVMS Alpha systems. For new program development for OpenVMS Alpha, Digital recommends the use of mid- and high-level languages. For more information on the MACRO–32 compiler, see *Porting VAX MACRO Code from OpenVMS VAX to OpenVMS Alpha*.

4.2.1.3 Other Development Tools

Several other tools in addition to the compilers are available to create native Alpha images:

- OpenVMS Linker

  The OpenVMS Linker can now accept VAX object files or Alpha object files to produce either a VAX image or an Alpha image. It also functions as a cross linker, since it can produce Alpha images while running on VAX hardware.

- OpenVMS Debugger

  The OpenVMS Debugger running on OpenVMS Alpha has the same command interface as the current OpenVMS VAX debugger. The graphical interface on OpenVMS VAX systems is also available on OpenVMS Alpha systems.

- OpenVMS Librarian utility

  The OpenVMS Librarian utility creates either VAX or Alpha libraries.

- OpenVMS Message utility

  The OpenVMS Message utility allows you to supplement the OpenVMS system messages with your own messages.

- MACRO–64 Assembler for OpenVMS Alpha

  The MACRO–64 assembler for OpenVMS Alpha systems is the native assembler for all Alpha computers. Unlike the VAX MACRO assembler, which is included with the OpenVMS VAX operating system, the MACRO–64 assembler is not included with the OpenVMS Alpha operating system. It can be purchased separately. In general, the mid- and high-level language compilers generate higher performance code for OpenVMS Alpha systems than the MACRO–64 assembler. Therefore, Digital recommends you use mid- and high-level compilers for new program development for OpenVMS Alpha systems. For more information about the MACRO–64 assembler, see the *MACRO–64 Assembler for OpenVMS AXP Systems Reference Manual*.

- ANALYZE/IMAGE

  The Analyze/Image utility can analyze either VAX or Alpha images.

- ANALYZE/OBJECT

  The Analyze/Object utility can analyze either VAX or Alpha objects.

- DECset

  DECset, a comprehensive set of CASE tools, includes the Language Sensitive Editor (LSE), Source Code Analyzer (SCA), Code Management System (CMS), Module Management System (MMS), and other components.

4.2.2 Translating

The process of translating a VAX image to run on an Alpha system is described in detail in *DECmigrate for OpenVMS AXP Systems Translating Images*. In general, the process is straightforward, although you may have to modify your code somewhat to get it to translate without error.

4.2.2.1 VAX Environment Software Translator (VEST) and Translated Image Environment (TIE)

The main tools for migrating VAX user-mode images to OpenVMS Alpha are a static translator and a run-time support environment:

- The VAX Environment Software Translator (VEST) is a utility that analyzes a VAX image and creates a functionally equivalent translated image. Using VEST, you will be able to do the following:
  - Determine whether a VAX image is translatable.
  - Translate the VAX image to an Alpha image.
  - Identify specific incompatibilities with OpenVMS Alpha within the image and, when appropriate, obtain information on how to correct those incompatibilities in the source files.
  - Identify ways to improve the run-time performance of the translated image.
- The Translated Image Environment (TIE) is an Alpha shareable image that supports translated images at run time. TIE provides the translated image with an environment similar to OpenVMS VAX and processes all interactions with the native Alpha system. Items that TIE provides include:
  - VAX instruction interpreter, which supports:
    - Execution of VAX instructions (including instruction atomicity) that is similar to their execution on a VAX system
    - Complex VAX instructions, as subroutines
  - VAX compatible exception handler
  - Jacket routines that allow communication between native and translated code
  - Emulated VAX stack

  TIE is invoked automatically for any translated image; you do not need to call it explicitly.

VEST locates and translates as much VAX code as possible into Alpha code. TIE interprets any VAX code that cannot be converted into Alpha instructions; for example:

- Instructions that VEST could not statically identify
- H_ and D56 (56-bit D_floating) floating-point operations

Since interpreting instructions is a slow process, requiring perhaps 100 Alpha instructions per average VAX instruction, VEST attempts to find and translate as much VAX code as possible to minimize the need for interpreting it at run time. A translated image runs at approximately one-third the speed of a comparable native Alpha image, depending on how much VAX code TIE needs to interpret. Translated VAX images run at least as fast as they would run on equivalent (same technology) VAX hardware.

Note that you cannot specify dynamic interpretation of a VAX image on an Alpha system. You must use VEST to translate the image before it can run on OpenVMS Alpha.

Translating a VAX image produces an image that runs as a native image on Alpha hardware. The Alpha image is not merely an interpreted or emulated version of the VAX image, but contains Alpha instructions that perform operations identical to those performed by the instructions in the original VAX image. The Alpha .EXE file also contains the original VAX image in its entirety, which allows TIE to interpret any code that VEST could not translate.

VEST's analysis capability also makes it useful for evaluating programs that you intend to recompile, rather than translate.

See *DECmigrate for OpenVMS AXP Systems Translating Images* for a complete description of VEST and TIE. The manual explains in detail all the output that VEST generates, such as flowgraphs, and how to interpret it. The manual also explains how information provided in image information files (IIFs) created by VEST can help you improve the translated image's run-time performance.

4.3 Debugging and Testing the Migrated Application

Once you have migrated your application to OpenVMS Alpha, you may have to debug it.

You will also need to test the application for correct operation.

4.3.1 Debugging

The OpenVMS operating system provides the following debuggers:

- The OpenVMS Debugger supports debugging of both VAX and native Alpha programs. This debugger does not support the debugging of translated images.

  The OpenVMS Debugger is a symbolic debugger, that is, the debugger allows you to refer to program locations by the symbols you used for them in your program—the names of variables, routines, labels, and so on. You do not need to specify memory addresses or machine registers when referring to program locations, although you can if you wish.

  Although the OpenVMS Debugger does not generally work for translated images, it is helpful in one area. Since the translated image mimics the VAX registers, you can use the commands SHOW CALLS and SHOW STATE to get some VAX context for more detailed debugging.

- The Delta Debugger supports debugging of VAX and Alpha programs. This debugger also supports the debugging of translated images.

  The Delta Debugger is an address location debugger, that is, the debugger requires you to refer to program locations by address location. This debugger is primarily used to debug programs that run in privileged processor mode or at an elevated interrupt level.

- The System-Code Debugger is a symbolic debugger that allows you to debug nonpageable code and device drivers running at any IPL.

- The Heap Analyzer provides a graphical representation of memory use in real time. This allows you to quickly identify inefficient memory usage in your application such as allocations that are made too often, memory blocks that are too large, fragmentation, or memory leaks.

Debugging must take place on Alpha hardware.

4.3.1.1 Debugging with the OpenVMS Debugger

On OpenVMS Alpha systems you can use the debugger with programs written in the following Digital languages:

- DEC Ada
- DEC BASIC
- DEC C
- DEC C++
- DEC COBOL
- DEC Fortran (VAX systems)
- Digital Fortran (Alpha systems)
- MACRO–32 (compiled with the MACRO-32 compiler)
- MACRO–64
- DEC Pascal
- DEC PL/I

The OpenVMS Debugger includes several features that address the architectural differences of OpenVMS Alpha code. These enable you to more easily debug code that you are porting to OpenVMS Alpha systems. For example, you can use the /UNALIGNED_DATA qualifier with the SET command to cause the debugger to break directly after any instruction that accesses unaligned data (such as a load word instruction which accesses data that is not on a word boundary).

You can use the /RETURN qualifier with the SET command for any routine. It is not limited to routines called with a CALLS or CALLG instruction as it is on an OpenVMS VAX system. For more information about features specific to OpenVMS Alpha systems, see the *OpenVMS Debugger Manual*.

When you debug your migrated application on an Alpha system with the OpenVMS Debugger, bear in mind the following considerations:

- You can use the debugger with programs written in any language for which there is an Alpha compiler available.
- The debugger does not support debugging of installed resident images. For more information on installed resident images, see the Bookreader version of the *OpenVMS System Manager's Manual: Tuning, Monitoring, and Complex Systems*.
- The debugger does not support debugging of inlined routines. If you attempt to debug an inlined routine, the debugger issues a message that it cannot access the routine:

  ```
  DBG> %DEBUG-E-ACCESSR, no read access to address 00000000
  ```

- The debugger does not completely support the debugging of Register Frame Procedures or No Frame Procedures. If you issue STEP/OVER or STEP /RETURN commands for these procedures, unexpected results can occur.

For more information on debugging with the OpenVMS Debugger, see the *OpenVMS Debugger Manual*.

4.3.1.2 Debugging with the Delta Debugger

The Delta/XDelta Debugger (DELTA/XDELTA), running on OpenVMS Alpha systems, provides enhancements to existing commands and several new commands necessitated by the Alpha architecture. The enhancements include the display of base registers in decimal instead of hexadecimal notation and the ability to look at the internal process identification (PID) number of another process. The new commands include ;Q, used to validate queues, and ;I, used to locate and display information about the current main image. For the Delta Debugger, the ;I command can also display information about all shareable images activated by the current main image. For more information about how the Delta/XDelta Debugger operates on OpenVMS Alpha systems, see the *OpenVMS Delta/XDelta Debugger Manual*.

You can use the Delta Debugger to debug applications that are partly or completely translated.

Translated Applications

When attempting to debug a translated image, you should:

- Make sure that the program you are translating works correctly under OpenVMS VAX Version 7.0.
- Make sure that VEST and any IIF files for run-time libraries are of the same release as the version of OpenVMS Alpha you are using.
- Use the VEST qualifiers /DEBUG, /LIST, and /SHOW=MACHINE_CODE to capture Alpha and VAX instructions. (Note that in the listing, an asterisk identifies a VAX instruction.) Have the VAX map and listing for the VAX image at hand for comparison.

Mixed Applications

To debug an application that is partly native Alpha code and partly translated code, make sure that the native parts of the application were compiled using the /TIE qualifier; in addition, you must link the application with the /NO_NATIVE_ONLY linker option.

For more information on debugging with the Delta Debugger, see the *OpenVMS Delta/XDelta Debugger Manual*.

For more information on debugging translated images, contact Systems Integration.

4.3.1.3 Debugging with the OpenVMS Alpha System-Code Debugger

The OpenVMS Alpha System-Code Debugger is available for debugging nonpageable system code and device drivers running at any IPL. The OpenVMS Alpha System-Code Debugger is a symbolic debugger. You can specify variable names, routine names, and so on, precisely as they appear in your source code. You can also display the source code where the software is executing and step through it by source line.

Note that running the System-Code Debugger requires two Alpha systems.

You can use this debugger to debug code written in the following languages:

- C
- BLISS

- VAX MACRO

Note

A BLISS compiler is available on the OpenVMS Freeware CD that ships with OpenVMS VAX Version 6.2 and OpenVMS Alpha Version 6.2.

The OpenVMS Alpha System-Code Debugger recognizes the syntax, data typing, operators, expressions, scoping rules, and other constructs of a given language. If your program is written in more than one language, you can change the debugging context from one language to another during a debugging session.

For more information about Step 2 drivers and the OpenVMS Alpha System-Code Debugger, see the *OpenVMS Alpha Device Support: Developer's Guide*.

4.3.2 Analyzing System Crashes

OpenVMS provides two tools for analyzing system crashes: the System Dump Analyzer and the Crash Log Utility Extractor.

4.3.2.1 System Dump Analyzer

The System Dump Analyzer (SDA) utility on OpenVMS Alpha systems is almost identical to the utility provided on OpenVMS VAX systems. Many commands, qualifiers, and displays are identical, although there are some additional commands and qualifiers, including several for accessing functions of the Crash Log Utility Extractor (CLUE) utility. Some displays have been adapted to show information specific to OpenVMS Alpha systems, such as processor registers and data structures.

While the SDA interface has changed only slightly, the contents of VAX and Alpha dump files and the entire process of analyzing a system crash from a dump differ significantly between the two computers. The Alpha execution paths leave more complex structures and patterns on the stack than VAX execution paths do.

To use SDA on a VAX computer, you must first familiarize yourself with the OpenVMS calling standard for VAX systems. Similarly, to use SDA on an Alpha system, you must familiarize yourself with the OpenVMS calling standard for Alpha systems before you can decipher the pattern of a crash on the stack.

The changes to SDA include the following:

– The displays of the SHOW CRASH and SHOW STACK commands contain additional information that make debugging fatal system exception bugchecks simpler.

– The SHOW EXEC command display includes additional information about executive images if they were loaded using image **slicing**. Slicing is a function performed by the executive image loader for executive images and by the OpenVMS Install utility for user-mode images. Slicing an executive image (or a user-mode image) greatly improves performance by reducing contention for the limited number of translation buffer entries.

– The MAP command, a new SDA command, enables you to map an address in memory to an image offset in a map file.

– A new symbol, FPCR, has been added to the symbol table. This symbol represents a floating-point register.

4.3.2.2 Crash Log Utility Extractor

The Crash Log Utility Extractor (CLUE) is a tool for recording a history of crash dumps and key parameters for each crash dump and for extracting and summarizing key information. Unlike crash dumps, which are overwritten with each system crash and are available only for the most recent crash, the crash history file (on OpenVMS VAX) and the summary crash history file with a separate listing file for each crash (on OpenVMS Alpha), are permanent records of system crashes.

The implementation differences between OpenVMS VAX and OpenVMS Alpha are shown in Table 4–1.

Table 4–1 CLUE Differences Between OpenVMS VAX and OpenVMS Alpha

Attribute	OpenVMS VAX	OpenVMS Alpha
Access method	Invoked as a separate utility.	Accessed through SDA.
History file	A cumulative file that contains a one-line summary and detailed information from the crash dump file for each crash.	A cumulative file that contains only a one-line summary for each crash dump. The detailed information for each crash is put in a separate listing file.
Uses in addition to debugging crash dumps	None.	CLUE commands can be used interactively to examine a running system.
Documentation	Bookreader versions of the *OpenVMS System Manager's Manual* and *OpenVMS System Management Utilities Reference Manual*	Bookreader versions of the *OpenVMS System Manager's Manual* and *OpenVMS Alpha System Dump Analyzer Utility Manual*

4.3.3 Testing

You must test your application to compare the performance and functionality of the migrated version with those of the original VAX version.

The first step in testing is to establish baseline values for your application by running your test suite on the VAX application.

Once your application is running on an Alpha system, there are two types of tests you will want to apply:

- The standard tests used for the VAX version of the application
- New tests to check specifically for problems due to the change in architecture

4.3.3.1 VAX Tests

Because the changes in your application are combined with use of a new architecture, testing your application after it is migrated to OpenVMS Alpha is particularly important. Not only can the changes introduce errors into the application, but the new environment may bring out latent problems in the VAX version.

Testing your migrated application involves the following steps:

1. Get a complete set of standard data for the application prior to the migration.
2. Migrate your test suite along with the application (if the tests are not already available on Alpha).

3. Validate the test suite on an Alpha system.
4. Run the migrated tests on the migrated application.

Both regression tests and stress tests are useful here. Stress tests are important to test for platform differences in synchronization, particularly for applications that use multiple threads of execution.

4.3.3.2 Alpha Tests

While your standard tests should go a long way toward verifying the function of the migrated application, you should add some tests that look at issues specific to the migration. Points to focus on include the following:

- Compiler differences—changes in optimization and data alignment
- Architectural differences—changes in instruction atomicity, memory atomicity, and read/write ordering, for example
- Integration—modules written in different languages, or modules that had to be translated

4.3.4 Uncovering Latent Bugs

Despite your best efforts, and following all the previous suggestions, you may encounter bugs that were in your program all along, but never caused a problem on an OpenVMS VAX system. For example, a failure to initialize some variable in your program might have been benign on a VAX computer but could produce an arithmetic exception on an Alpha computer. The same could be true moving between any other two architectures, because the available instructions and the way compilers optimize them is bound to change. There is no magic answer for bugs that have been in hiding, but you should test your programs after porting them before making them available to other users.

4.4 Integrating the Migrated Application into a Software System

After you have migrated your application by recompiling or translating it, check for problems that are caused by interactions with other software and that may have been introduced during the migration.

Sources of problems in interoperability can include the following:

- Alpha and VAX systems within a VMScluster environment must use separate system disks. You must make sure that your application refers to the appropriate system disk.
- Image names

  In a mixed environment, be sure that your application refers to the correct version.

  - Native VAX and native Alpha versions of an image have the same name.
  - The translated version of an image has the string "_TV" added to its name.
- Recompiled images may expect naturally aligned data, while translated images have unaligned data, like the original VAX image.

5
Recompiling and Relinking Overview

This chapter introduces the general process of moving an application that runs on a VAX system to an Alpha system by recompiling and relinking the source files that make up the application. Specifically, this chapter covers the following topics:

- Using Alpha versions of tools, such as native compilers and the linker
- Identifying dependencies your application may have on aspects of the VAX architecture

5.1 Overview

In general, if your application is written in a high-level programming language, you should be able to get it running on an Alpha system with a minimum of effort. High-level languages insulate applications from dependence on the underlying machine architecture. In addition, for the most part, the programming environment on Alpha systems duplicates the programming environment on VAX systems. Using native Alpha versions of the language compilers and the OpenVMS Linker utility (linker), you can recompile and relink the source files that make up your application to produce a native Alpha image.

However, it is possible to introduce architectural dependencies even in applications written in high-level languages. The following sections describe the programming environment on an Alpha system and provide guidelines for identifying code in your application source files that may not be able to be moved to an Alpha system without modification.

5.2 Recompiling Your Application with Native Alpha Compilers

Many of the languages supported on VAX systems, such as FORTRAN and C, are also supported on Alpha systems. For complete information about the availability of programming languages on Alpha systems, see Chapter 12.

The compilers available on Alpha systems are intended to be compatible with their counterparts on VAX systems. The compilers conform to language standards and include support for most VAX language extensions. The compilers produce output files with the same default file types as they do on VAX systems, such as .OBJ for an object module.

Note, however, that some features supported by the compilers on VAX systems may not be available in the same compiler on Alpha systems. In addition, some compilers on Alpha systems support new features not supported by their counterparts on VAX systems. To provide compatibility, some compilers support compatibility modes. For example, the DEC C for OpenVMS Alpha systems compiler supports a VAX C compatibility mode that is invoked by specifying the /STANDARD=VAXC qualifier. Chapter 12 lists the features of several compilers available on both the VAX and Alpha systems.

5.3 Relinking Your Application on an Alpha System

Once you successfully recompile your source files, you must relink your application to create a native Alpha image. The linker produces output files with the same file types as on current VAX systems. For example, by default, the linker uses the file type .EXE to identify image files.

Because the way in which you perform certain linking tasks is different on Alpha systems, you will probably need to modify the LINK command used to build your application. The following list describes some of these linker changes that may affect your application's build procedure. See the Bookreader version of the *OpenVMS Linker Utility Manual* for more information.

- **Declaring universal symbols in shareable images**—If your application creates shareable images, your application build procedure probably includes a transfer vector file, written in VAX MACRO, in which you declare the universal symbols in the shareable image. On Alpha systems, instead of creating a transfer vector file, you must declare universal symbols in a linker options file by specifying the SYMBOL_VECTOR= option.
- **Linking against the OpenVMS executive**—On VAX systems, you link against the OpenVMS executive by including the system symbol table file (SYS.STB) in your build procedure. On Alpha systems, you link against the OpenVMS executive by specifying the /SYSEXE qualifier.
- **Optimizing the performance of images**—On Alpha systems, the linker can perform certain optimizations that can improve the performance of the images it creates. The linker can also enhance performance by creating shareable images that can be installed as resident images.
- **Processing shareable images implicitly**—On VAX systems, when you specify a shareable image in a link operation, the linker also processes all the shareable images to which that shareable image was linked. On Alpha systems, you must specify these shareable images to include them in your build procedure.

The linker supports several qualifiers and options, listed in Table 5–1, that are specific to Alpha systems. Table 5–2 lists linker qualifiers supported on VAX systems but not on Alpha systems.

Table 5–1 Linker Qualifiers and Options Specific to OpenVMS Alpha Systems

Qualifiers	Description
/DEMAND_ZERO	Controls how the linker creates demand-zero image sections.
/DSF	Directs the linker to create a file called a debug symbol file (DSF) for use by the OpenVMS Alpha System-Code Debugger.
/GST	Directs the linker to create a global symbol table (GST) for a shareable image (the default). More typically specified as /NOGST when used to create shareable images for run-time kits.

(continued on next page)

Table 5–1 (Cont.) Linker Qualifiers and Options Specific to OpenVMS Alpha Systems

Qualifiers	Description
/INFORMATIONALS	Directs the linker to output informational messages during a link operation (the default). More typically specified as /NOINFORMATIONALS to suppress these messages.
/NATIVE_ONLY	Directs the linker to *not* pass along the procedure signature block (PSB) information, created by the compilers, in the image it is creating (the default). If /NONATIVE_ONLY is specified during linking, the image activator uses the PSB information, if any, provided in the object modules specified as input files to the link operation to invoke jacket routines. Jacket routines are necessary to allow native Alpha images to work with translated VAX images.
/REPLACE	Directs the linker to perform certain optimizations that can improve the performance of the image it is creating, when requested to do so by the compilers (the default).
/SECTION_BINDING	Directs the linker to create a shareable image that can be installed as a resident image.
/SYSEXE	Directs the linker to process the OpenVMS executive image (SYS$BASE_IMAGE.EXE) to resolve symbols left unresolved in a link operation.

Options	Description
SYMBOL_TABLE= option	Directs the linker to include global symbols as well as universal symbols in the symbol table file associated with a shareable image. By default, the linker includes only universal symbols.
SYMBOL_VECTOR= option	Used to declare universal symbols in Alpha shareable images.

Table 5–2 Linker Options Specific to OpenVMS VAX Systems

Options	Description
BASE= option	Specifies the base address (starting address) that you want the linker to assign to the image.
DZRO_MIN= option	Specifies the minimum number of contiguous, uninitialized pages that the linker must find in an image section before it can extract the pages from the image section and place them in a newly created demand-zero image section. By creating demand-zero image sections (image sections that do not contain initialized data), the linker can reduce the size of images.
ISD_MAX= option	Specifies the maximum number of image sections allowed in the image.
UNIVERSAL= option	Declares a symbol in a shareable image as universal, causing the linker to include it in the global symbol table of a shareable image.

5.4 Compatibility Between the Mathematics Libraries Available on VAX and Alpha Systems

Mathematical applications using the standard OpenVMS call interface to the OpenVMS Mathematics (MTH$) Run-Time Library need not change their calls to MTH$ routines when migrating to an OpenVMS Alpha system. Jacket routines are provided that map MTH$ routines to their math$ counterparts in the Digital Portable Mathematics Library (DPML) for OpenVMS Alpha systems. However, there is no support in the DPML for calls made to JSB entry points and vector routines. Note that DPML routines are different from those in the OpenVMS MTH$ RTL and you should expect to see small differences in the precision of the mathematical results.

To maintain compatibility with future libraries and to create portable mathematical applications, Digital recommends that you use the DPML routines available through the high-level language of your choice (for example, DEC C or DEC Fortran) rather than using the call interface. Significantly higher performance and accuracy are also available to you with DPML routines.

See the *Digital Portable Mathematics Library* manual for more information about the DPML routines.

5.5 Determining the Host Architecture

Your application may need to determine whether it is running on an OpenVMS VAX system or an Alpha system. From within your program, you can obtain this information by calling the $GETSYI system service (or the LIB$GETSYI RTL routine), specifying the ARCH_TYPE item code. When your application is running on a VAX system, the $GETSYI system service returns the value 1. When your application is running on an Alpha system, the $GETSYI system service returns the value 2.

Example 5–1 shows how to determine the host architecture in a DCL command procedure by calling the F$GETSYI DCL command and specifying the ARCH_TYPE item code. (For an example of calling the $GETSYI system service to obtain the page size of an Alpha system, see Section 6.4.)

Example 5–1 Using the ARCH_TYPE Keyword to Determine Architecture Type

```
$! Determine architecture type
$ type_symbol = f$getsyi("arch_type")
$ if type_symbol .eq. 1 then goto ON_VAX
$ if type_symbol .eq. 2 then goto ON_ALPHA
$ ON_VAX:
$ !
$ ! Do VAX-specific processing
$ !
$ exit
$ ON_ALPHA:
$ !
$ ! Do Alpha-specific processing
$ !
$ exit
```

Note, however, that the ARCH_TYPE item code is available only on VAX systems running OpenVMS Version 5.5 or later. If your application needs to determine the host architecture for earlier versions of the operating system, use one of the other $GETSYI system service item codes listed in Table 5–3.

Table 5–3 $GETSYI Item Codes That Specify Host Architecture

Keyword	Usage
ARCH_TYPE	Returns 1 on a VAX system; returns 2 on an Alpha system. Supported on Alpha systems and on VAX systems running OpenVMS Version 5.5 or later.
ARCH_NAME	Returns text string "VAX" on VAX systems and text string "Alpha" on Alpha systems. Supported on Alpha systems and on VAX systems running OpenVMS Version 5.5 or later.
HW_MODEL	Returns an integer that identifies a particular hardware model. All values equal to or larger than 1024 identify Alpha systems.
CPU	Returns an integer that identifies a particular CPU. The value 128 identifies a system as "not a VAX." This code is supported on much earlier versions of OpenVMS than the ARCH_TYPE and ARCH_NAME codes.

6
Adapting Applications to a Larger Page Size

This chapter describes how to identify dependencies your application may have on the VAX page size and makes recommendations for correcting those dependencies.

6.1 Overview

In general, page size, the basic unit of memory manipulated by the operating system, is below the level of applications, especially for applications written in high- or mid-level programming languages. However, your application may contain page-size dependencies if it calls system services or run-time library routines to perform memory management functions such as the following:

- Allocating virtual memory
- Mapping sections into the virtual address space of your process
- Locking memory into your working set
- Protecting segments of your virtual address space

The system services and run-time library routines that perform these functions manipulate memory in pages. The values you specified as arguments to these routines are based on an assumption of a 512-byte page, the page size defined by the VAX architecture. The Alpha architecture supports an 8K, 16K, 32K, or 64K byte page size, depending on the implementation, so you should examine the values you specify as arguments to the routines to make sure they still satisfy the requirements of your application. The following sections provide more information about examining the routines.

Note that this difference in page sizes does not affect memory allocation using higher level routines, for example, the run-time library routines that manipulate virtual memory zones or language-specific memory allocation routines such as the `malloc` and `free` routines in C.

6.1.1 Compatibility Features

Wherever possible, system services or run-time library routines attempt to present the same interface and return values on Alpha systems as they do on VAX systems. For example, on Alpha systems, the routines that accept page-count values as arguments still interpret these arguments in 512-byte quantities, now called **pagelets** to distinguish them from the CPU-specific page size. The routines convert pagelet values into CPU-specific pages. The routines that return page-count values convert from CPU-specific pages to pagelets so that the return values expected by your application are still measured in 512-byte units.

Note

On Alpha systems, when creating page frame sections using the $CRMPSC system service (with the SEC$M_PFNMAP flag bit set), the value specified in the page count argument (**pagcnt**) is interpreted as the CPU-specific page size, *not* as a pagelet value.

6.1.2 Summary of Memory Management Routines with Potential Page-Size Dependencies

Despite the compatibility, some routines behave differently on Alpha systems than they do on VAX systems and may require you to modify your source code. For example, on Alpha systems, the system services that map section files ($CRMPSC and $MGBLSC) require you to specify address value arguments that are aligned on CPU-specific page boundaries. On VAX systems, these routines round the address values specified in arguments to VAX page boundaries. On Alpha systems, the routines do not round these addresses to CPU-specific page boundaries.

Table 6–1 lists the memory management routines with the arguments they support that may contain page-size dependencies. The table lists the arguments with their intended function and describes how these arguments are interpreted on Alpha systems. Note that the table does not attempt to list all the arguments accepted by each routine. For more information about the routines and their argument lists, see the *OpenVMS System Services Reference Manual.*

Table 6–1 Potential Page-Size Dependencies in Memory Management Routines

Routine	Argument	Behavior on Alpha Systems
Adjust Working Set Limit ($ADJWSL)	**pagcnt** specifies the number of pages to add to (or subtract from) the current working set limit.	Interpreted in pagelets, adjusted up or down to represent CPU-specific-sized pages.
	wsetlm specifies the value of the current working set limit.	Interpreted in pagelets, adjusted up or down to represent CPU-specific-sized pages.
Create Process ($CREPRC)	**quota** accepts several quota descriptors that specify page counts, such as the default working set size, paging file quota, and working set expansion quota.	Interpreted in pagelets, adjusted up or down to represent CPU-specific-sized pages.
Create Virtual Address ($CRETVA)	**inadr** specifies the start- and end-addresses of the memory to be allocated. If the end-address is the same as the start-address, a single page is allocated.	Addresses are adjusted up or down to fall on CPU-specific page boundaries.
	retadr specifies the actual start- and end-addresses of the memory affected by the call.	Unchanged.

(continued on next page)

Table 6–1 (Cont.) Potential Page-Size Dependencies in Memory Management Routines

Routine	Argument	Behavior on Alpha Systems
Create and Map Section ($CRMPSC)	**inadr** specifies the start- and end-addresses that define the region to be remapped. If the end-address is the same as the start-address, a single page is mapped, unless the SEC$M_EXPREG flag is set, in which case the start-address is interpreted as determining whether the allocation should be in P0 or P1 space.	Addresses must be aligned on CPU-specific pages (unless the SEC$M_EXPREG flag is set); no rounding is done. (See Section 6.3 for more information about mapping.)
	retadr specifies the actual start- and end-addresses of the memory affected by the call.	Returns the start- and end-addresses of the *usable* range of addresses, which may be different than the total amount mapped. This argument is required when the **relpag** argument is specified.
	flags specifies the type and characteristics of the section to be created or mapped.	The flag bit SEC$M_NO_OVERMAP indicates that existing address space should not be overmapped. When the flag bit SEC$M_PFNMAP is set, the **pagcnt** argument is interpreted as CPU-specific pages, *not* pagelets.
	relpag specifies the page offset at which mapping of the section file should begin.	Interpreted as an index into the section file, measured in pagelets.
	pagcnt specifies the number of pages (blocks) in the file to be mapped.	Interpreted in pagelets; no rounding is done. When the flag bit SEC$M_PFNMAP is set, the **pagcnt** argument is interpreted as CPU-specific pages, *not* pagelets.
	pfc specifies the number of pages that should be mapped when a page fault occurs.	Interpreted in CPU-specific-sized pages. When specifying a value for this argument, remember that, because Alpha systems support 8K, 16K, 32K, and 64K byte physical page sizes, at least 16 pagelets will be mapped for each physical page. The system cannot map less than a physical page.
Delete Virtual Address ($DELTVA)	**inadr** specifies the start- and end-addresses of the memory to be deallocated.	Addresses are adjusted up or down to fall on CPU-specific page boundaries.
	retadr specifies the actual start- and end-addresses of the memory that was deleted.	Unchanged.

(continued on next page)

Table 6–1 (Cont.) Potential Page-Size Dependencies in Memory Management Routines

Routine	Argument	Behavior on Alpha Systems
Expand Program/Control Region ($EXPREG)	**pagcnt** specifies the amount of memory to allocate, in 512-byte units.	Interpreted in pagelets.
	retadr specifies the actual start- and end-addresses of the memory affected by the call.	Unchanged.
Get Job/Process Information ($GETJPI)	**itmlst** specifies which information about the process is to be returned.	Many items, such as JPI$_WSEXTENT, interpreted as pagelet values. See the *OpenVMS System Services Reference Manual* for more information.
Get Queue Information ($GETQUI)	**itmlst** specifies information to be used in performing the function specified by the **func** argument.	Several items interpreted as pagelet values. See the *OpenVMS System Services Reference Manual* for more information.
Get Systemwide Information ($GETSYI)	**itmlst** specifies which information is to be returned about the node or nodes.	Several items interpreted as pagelet values. One additional item, SYI$_PAGE_SIZE, specifies the page size supported by the node. See the *OpenVMS System Services Reference Manual* for more information.
Get User Authorization Information ($GETUAI)	**itmlst** specifies which information from the user's user authorization file is to be returned.	Several items return pagelet values. See the *OpenVMS System Services Reference Manual* for more information.
Lock Page ($LCKPAG)	**inadr** specifies the start- and end-addresses of the memory to be locked.	Addresses are adjusted up or down to fall on CPU-specific page boundaries.
	retadr specifies the actual start- and end-addresses of the memory that was locked.	Unchanged.
Lock Working Set ($LKWSET)	**inadr** specifies the start- and end-addresses of the memory to be locked.	Addresses are adjusted up or down to fall on CPU-specific page boundaries.
	retadr specifies the actual start- and end-addresses of the memory that was locked.	Unchanged.
Map Global Section ($MGBLSC)	**inadr** specifies the start- and end-addresses that define the region to be remapped. If the end-address is the same as the start-address, a single page is mapped, unless the SEC$M_EXPREG flag is set, in which case the start-address is interpreted as determining whether the allocation should be in P0 or P1 space.	Addresses must be aligned on a CPU-specific page (unless the SEC$M_EXPREG flag is set); no rounding is done. (See Section 6.3 for more information about mapping.)

(continued on next page)

Table 6–1 (Cont.) Potential Page-Size Dependencies in Memory Management Routines

Routine	Argument	Behavior on Alpha Systems
	retadr specifies the actual start- and end-addresses of the memory affected by the call.	Returns start- and end-addresses of *usable* portion of memory mapped.
	relpag specifies the page offset at which mapping of the section file should begin.	Interpreted as an index into the section file, measured in pagelets.
Purge Working Set ($PURGWS)	**inadr** specifies the start- and end-addresses of the memory to be purged.	Addresses are adjusted up or down to fall on CPU-specific page boundaries.
Set Protection ($SETPRT)	**inadr** specifies the start- and end-addresses of the memory to be protected.	Addresses are adjusted up or down to fall on CPU-specific page boundaries.
	retadr specifies the actual start- and end-addresses of the memory that was protected.	Unchanged.
Set User Authorization File ($SETUAI)	**itmlst** specifies which information from the user authorization file is to be set.	Several items interpreted in pagelet values. See the *OpenVMS System Services Reference Manual* for more information.
Send to Job Controller ($SNDJBC)	**itmlst** specifies information to be used in performing the function specified by the **func** argument.	Several items interpreted in pagelet values. See the *OpenVMS System Services Reference Manual* for more information.
Unlock Page ($ULKPAG)	**inadr** specifies the start- and end-addresses of the memory to be unlocked.	Addresses are adjusted up or down to fall on CPU-specific page boundaries.
	retadr specifies the actual start- and end-addresses of the memory that was unlocked.	Unchanged.
Unlock Working Set ($ULWSET)	**inadr** specifies the start- and end-addresses of the memory to be unlocked.	Addresses are adjusted up or down to fall on CPU-specific page boundaries.
	retadr specifies the actual start- and end-addresses of the memory that was unlocked.	Unchanged.
Update Section ($UPDSEC)	**inadr** specifies the start- and end-address of the section to write to disk.	Rounds requests to CPU-specific pages. Note that only the address range actually represented by on-disk storage will be written to disk.
	retadr specifies the actual start- and end-addresses of the memory that was written to disk.	Addresses are adjusted up or down to fall on CPU-specific page boundaries.

The run-time library routines listed in Table 6–2 allocate (or free) pages of memory. For compatibility, these routines also interpret the page-count information you specify in pagelets.

Table 6–2 Potential Page-Size Dependencies in Run-Time Library Routines

Routine	Argument	Behavior on Alpha Systems
LIB$GET_VM_PAGE	**number-of-pages** argument specifies the number of contiguous pages to allocate.	Interpreted in pagelets, rounded to CPU-specific pages.
LIB$FREE_VM_PAGE	**number-of-pages** argument specifies the number of contiguous pages to free.	Interpreted in pagelets, rounded to CPU-specific pages.

6.2 Examining Memory Allocation Routines

To determine if the memory allocation performed by your application requires modification, check to see where the memory is allocated. The system service routines that perform memory allocation ($EXPREG and $CRETVA) allow you to allocate memory in two ways:

- By expanding the size of the P0 or P1 regions of your application's virtual address space
- By reclaiming a region of your application's existing virtual address space, starting at a location you specify

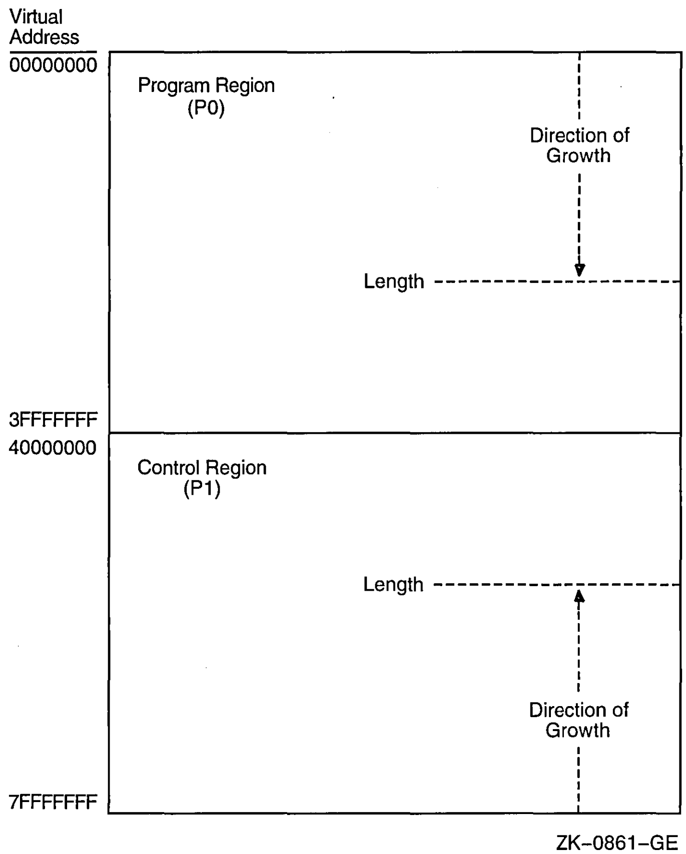
The Alpha architecture defines the same virtual address space layout as the VAX architecture and allows for growth of the P0 and P1 regions in the same direction as on VAX systems. Figure 6–1 shows this layout.

6.2.1 Allocating Memory in Expanded Virtual Address Space

If your application allocates memory by *expanding* virtual address space using the $EXPREG system service, you may not need to make any source code changes because the values you specified as arguments are valid on Alpha systems and VAX systems. The reasons for this are as follows:

- On Alpha systems, the $EXPREG system service interprets the amount of memory requested (specified as a page count in the **pagcnt** argument) in 512-byte units, the same as on an VAX system. Thus, the value your application specified still requests the same amount of memory. Note, however, that because the system service rounds the value up to CPU-specific pages, the actual amount of memory allocated by the system for your application may be larger on an Alpha system than it is on a VAX system. The entire amount of memory allocated is available for use by your application. Because applications typically allocate memory to satisfy buffer requirements, which do not change with different platforms, the value you specified should still satisfy the requirements of your application.
- Because the allocation occurs in an expanded area of virtual address space, the discrepancy between the amount requested and the amount actually allocated by the system should have no effect on the function of your application.

Figure 6–1 Virtual Address Layout



Recommendation

Your application may not need to be modified. However, Digital suggests that you obtain the exact boundaries of the memory allocated by the system, because the amount of memory returned by the $EXPREG system service may vary among implementations of the Alpha architecture. To do this, specify the optional **retadr** argument to the $EXPREG system service, if your application does not already include it. The **retadr** argument contains the start-address and the end-address of the memory allocated by the system service.

For example, the program in Example 6–1 calls the $EXPREG system service to request 10 additional pages of memory. If you run this program on a VAX system, the $EXPREG system service allocates 5120 bytes of additional memory. If you run this program on an Alpha system, the $EXPREG system service allocates at least 8192 bytes and possibly more, depending on the page size of the particular implementation of the Alpha architecture.

Example 6–1 Allocating Memory by Expanding Your Virtual Address Space

```
#include  <ssdef.h>
#include  <stdio.h>
#include  <stsdef.h>
#include  <descrip.h>
#include  <dvidef.h>

#define  PAGE_COUNT 10  ❶
#define  P0_SPACE   0
#define  P1_SPACE   1

main( argc, argv )
int argc;
char *argv[];
{
     int    status = 0;
     long   bytes_allocated, addr_returned[2];

❷    status = SYS$EXPREG( PAGE_COUNT, &addr_returned, 0, P0_SPACE);

     bytes_allocated = addr_returned[1] - addr_returned[0];

     if( status == SS$_NORMAL)
        printf("bytes allocated = %d\n", bytes_allocated );
     else
        return (status);

}
```

The items in the following list correspond to the numbered items in Example 6–1:

❶ The example defines a symbol, PAGE_COUNT, to stand for the number of pages requested.

❷ The example requests 10 additional pages to be added at the end of the P0 region of its virtual address space.

6.2.2 Allocating Memory in Existing Virtual Address Space

If your application reallocates memory that is already in its virtual address space by using the $CRETVA system service, you may need to modify the values of the following arguments to $CRETVA:

- If your application explicitly rounds the address specified in the **inadr** argument to be a multiple of 512 in order to align on a VAX page boundary, you need to modify the address. On Alpha systems, the $CRETVA system service rounds the start-address down to a CPU-specific page boundary, which will vary with different implementations.
- The size of the reallocation, specified by the address range in the **inadr** argument, may be larger on an Alpha system than it is on a VAX system because the request is rounded up to CPU-specific pages. This can cause the unintended destruction of neighboring data, which also occurs with single-page allocations. (When the start-address and the end-address specified in the **inadr** argument match, a single page is allocated.)

Recommendations

To determine whether your application needs to be modified, Digital suggests doing the following:

- For all potential page sizes, make sure the area of virtual address space affected by the call does not destroy important data.

- For all potential page sizes, make sure the start-address at which the allocation begins always falls on a page boundary.
- Specify the optional **retadr** argument, if not already included by your application, to determine the exact boundaries of the memory allocated by the call to the $CRETVA system service.

Example 6–2 shows how memory allocated to a buffer can be reallocated by using the $CRETVA system service.

Example 6–2 Allocating Memory in Existing Address Space

```
#include  <ssdef.h>
#include  <stdio.h>
#include  <stsdef.h>
#include  <descrip.h>
#include  <dvidef.h>

char _align(page) buffer[1024];

main( argc, argv )
int argc;
char *argv[];
{
     int      status = 0;
     long     inadr[2];
     long     retadr[2];

     inadr[0] = &buffer[0];
     inadr[1] = &buffer[1023];

     printf("inadr[0]=%u,inadr[1]=%u\n",inadr[0],inadr[1]);

     status = SYS$CRETVA(inadr, &retadr, 0);

     if( status & STS$M_SUCCESS )
     {
        printf("success\n");
        printf("retadr[0]=%u,retadr[1]=%u\n",retadr[0],retadr[1]);
     }
     else
     {
        printf("failure\n");
        exit(status);
     }
}
```

6.2.3 Deleting Virtual Memory

Calls to the $DELTVA system service to free memory allocated by the $EXPREG and $CRETVA system services should require no modification if your application uses the address range returned in the **retadr** argument (returned by the routine used to allocate the memory) as the **inadr** argument to the $DELTVA system service. Because the actual amount of the allocation will vary with the implementation, your application should not make any assumptions regarding the extent of the allocation.

6.3 Examining Memory Mapping Routines

To determine if the memory mapping performed by your application requires modification, check to see where in virtual memory your application performs the mapping. The memory mapping system services ($CRMPSC and $MGBLSC) allow you to map memory in the following ways:

- Map memory into an expanded area of your application's virtual address space
- Map a single page of memory into your application's virtual address space, starting at a location you specify (the location may be in existing virtual address space)
- Map memory into an existing area of your virtual address space, defined by the start- and end-addresses you specify

How your application maps a section is determined primarily by the following arguments to the $CRMPSC and $MGBLSC system services:

- **inadr** argument—Specifies the size and location of the section by its start- and end-addresses, interpreted by the $CRMPSC system service in the following ways:
  - If both addresses specified in the **inadr** argument are the same and the SEC$M_EXPREG bit is set in the **flags** argument, the system service allocates the memory in whichever program region the addresses fall, but does not use the specified location.
  - If both addresses specified in the **inadr** argument are the same and the SEC$M_EXPREG flag is not set, a single page is mapped, starting at the specified location. (Note that this mode of operation of the $CRMPSC system service is not supported on Alpha systems. If your application uses this mode, see Section 6.3.2 for recommendations about modifying your source code.)
  - If both addresses are different, the system service maps the section into memory using the boundaries specified.
- **pagcnt** (page count) argument—Specifies the number of blocks you want to map from the section file.
- **relpag** (relative page number) argument—Specifies the location in the section file at which you want mapping to begin.

The $CRMPSC and $MGBLSC system services map a miminum of one CPU-specific page. If the section file does not fill a single page, the remainder of the page is filled with zeros. The extra space on the page should not be used by your application because only the data that fits into the section file will be written back to the disk.

6.3.1 Mapping into Expanded Virtual Address Space

If your application maps a section file into an expanded area of your application's virtual address space, you may not need to modify the source code. Because the mapping occurs in expanded virtual address space, there is no danger of overmapping existing data, even if the amount of memory allocated is larger on an Alpha system than on a VAX system. Thus, the values you specify as arguments to the $CRMPSC system service on a VAX system should still work on an Alpha system.

Recommendation

While applications that map sections into expanded areas of virtual memory may work correctly without modification, Digital suggests that you specify the **retadr** argument, if not already specified by your application, to determine the exact boundaries of the memory that was mapped by the call.

Note

If your application specifies the **relpag** argument, you must specify the **retadr** argument; it is not an optional argument. For more information about using the **relpag** argument, see Section 6.3.4.

Example 6–3 shows a call to the $CRMPSC system service that maps a section file into expanded address space. The example maps a section file named MAPTEST.DAT that was created using the DCL CREATE command, as follows:

```
$ CREATE maptest.dat
test data test data test data test data test data
test data test data test data test data test data
test data test data test data test data test data
test data test data test data test data test data
test data test data test data test data test data
test data test data test data test data test data
test data test data test data test data test data
test data test data test data test data test data
Ctrl/Z
```

Example 6–3 Mapping a Section into Expanded Virtual Address Space

```
#include  <ssdef.h>
#include  <string.h>
#include  <stdlib.h>
#include  <stdio.h>
#include  <stsdef.h>
#include  <descrip.h>
#include  <dvidef.h>
#include  <rms.h>
#include  <secdef.h>

struct FAB fab;

char _align(page) buffer[1024];
char *filename = "maptest.dat";

main( argc, argv )
int argc;
char *argv[];
{
     int    status = 0;
     long   flags = SEC$M_EXPREG;
     long   inadr[2];
     long   retadr[2];
     int    fileChannel;
```

(continued on next page)

Example 6–3 (Cont.) Mapping a Section into Expanded Virtual Address Space

```
/********  create disk file to be mapped *************/

     fab = cc$rms_fab;
     fab.fab$l_fna = filename;
     fab.fab$b_fns = strlen( filename );
     fab.fab$l_fop = FAB$M_CIF | FAB$M_UFO;  /* must be UFO */

     status = sys$create( &fab );

     if( status & STS$M_SUCCESS )
        printf("%s opened\n",filename);
     else
     {
        exit( status );
     }

     fileChannel = fab.fab$l_stv;

/**********  create and map the section  ****************/

     inadr[0] = &buffer[0];
     inadr[1] = &buffer[0];

     status = SYS$CRMPSC( inadr, /* inadr=address target for map */
                        &retadr, /* retadr= what was actually mapped */
                              0, /* acmode  */
                          flags, /* flags, with SEC$M_EXPREG bit set */
                              0, /* gsdnam, only for global sections */
                              0, /* ident, only for global sections */
                              0, /* relpag, only for global sections */
                    fileChannel, /* returned by SYS$CREATE */
                              0, /* pagcnt = size of sect. file used */
                              0, /* vbn = first block of file used */
                              0, /* prot = default okay */
                              0); /* page fault cluster size */

     if( status & STS$M_SUCCESS )
     {
          printf("section mapped\n");
          printf("retadr[0]=%u,retadr[1]=%u\n",retadr[0],retadr[1]);
     }
     else
     {
          printf("map failed\n");
          exit( status );
     }

}
```

6.3.2 Mapping a Single Page to a Specific Location

If your application maps a section file into a single page of memory, you will need to modify your source code because this mode of operation is not supported on Alpha systems. Because the page size on Alpha systems differs from that on VAX systems and varies with different implementations of the Alpha architecture, you must specify the exact boundaries of the memory into which you intend to map a section file. The $CRMPSC system service returns an invalid arguments error (SS$_INVARG) for this usage.

To see if your application uses this mode, check the start- and end-addresses specified in the **inadr** argument. If both addresses are the same *and* the SEC$M_EXPREG bit in the **flags** argument is *not* set, your application is using this mode.

Recommendations

Digital suggests the following guidelines when modifying calls to the $CRMPSC system service in this mode:

- If the location into which the mapping occurs is unimportant, set the SEC$M_EXPREG bit in the **flags** argument and let the system service map the section into an expanded area of your application's virtual address space. For more information about this mode of operation, see Section 6.3.1.
- If the location into which the mapping occurs is important, define both the start- and end-addresses in the **inadr** argument and map the section into a defined area. For more information about this mode, see Section 6.3.3.

6.3.3 Mapping into a Defined Address Range

If your application maps a section into a defined area of its virtual address space, you may need to modify your source code because, on Alpha systems, the $CRMPSC and $MGBLSC system services interpret some of the arguments differently than on VAX systems. The differences are as follows:

- The start-address specified in the **inadr** argument must be aligned on a CPU-specific page boundary and the end-address specified must be aligned with the end of a CPU-specific page. On VAX systems, the $CRMPSC and the $MGBLSC system services round these addresses to page boundaries for you. On Alpha systems, automatic rounding is not done because rounding to CPU-specific page boundaries affects a much larger portion of memory due to the larger page sizes on Alpha systems. Thus, on Alpha systems, you must explicitly state where you want the virtual memory space mapped. If the addresses you specify are not aligned on CPU-specific page boundaries, the $CRMPSC system service returns an invalid arguments error (SS$_INVARG).
- The addresses returned in the **retadr** argument reflect only the usable portion of the actual memory mapped by the call, not the entire amount mapped. The usable amount is either the value specified in the **pagcnt** argument (measured in pagelets) or the size of the section file, whichever is smaller. The actual amount mapped depends on how many CPU-specific pages are required to map the section file. If the section file does not fill a CPU-specific page, the remainder of the page is filled with zeros. The excess space on this page should not be used by your application. The end-address specified in the **retadr** argument specifies the upper limit available to your application. Note also that, when the **relpag** argument is specified, you must also include the **retadr** argument; it is not an optional argument on Alpha systems as it is on VAX systems. See Section 6.3.4 for more information.

Recommendations

Digital suggests that you change your application so that it maps data into expanded virtual address space, if possible. If you cannot change the way your application maps data, Digital recommends the following guidelines:

- Because the operating system maps a minimum of one physical page and physical pages on Alpha systems are larger than pages on VAX systems, you must make sure that when the system maps the section into the buffer you define in your application it does not overwrite neighboring data. Most applications on VAX systems define the buffer into which the section is to be mapped in multiples of 512 bytes because that is the page size on VAX systems, even if the section file to be mapped is less than 512 bytes in size. To follow this strategy on Alpha systems, you would need to declare a buffer in your application as large as the largest possible Alpha page, 64K bytes, which would waste memory.

A better way to make sure your section does not overwrite neighboring data when it is mapped is to force the linker to isolate the buffer into a separate image section. (The linker creates an image out of image sections. Each image section defines the memory requirements of part of the image.) By isolating the buffer into its own image section, you ensure that the mapping operation will not overwrite neighboring data because the linker allocates image sections on page boundaries; neighboring data will start on the next page boundary. Thus, you can map a page of memory into your section without disturbing neighboring data and without having to change the size of the buffer.

To ensure that the linker puts your section into its own image section, you must set the SOLITARY attribute of the program section in which your section resides, using the linker's PSECT_ATTR= option. (For more information, see the Bookreader version of the *OpenVMS Linker Utility Manual*.) Note that you may need to use the capabilities of whatever high- or mid-level programming language you are using to ensure that the compiler puts the buffer you define into a separate program section. See compiler documentation for more information.

- Make sure that the start- and end-addresses of the section that you specify as arguments to the $CRMPSC and $MGBLSC system services are aligned with the start- and end-addresses of a CPU-specific page. On VAX systems, the system services round the addresses to page boundaries for you. On Alpha systems, the system services do not round the addresses you specify to page boundaries.

  If you isolate the section into its own image section, using the SOLITARY program section attribute, the start-address is guaranteed to be on a page boundary because the linker aligns image sections on page boundaries by default, no matter what the page size of the host machine is at run time.

  To make sure the end-address of the section is aligned on a CPU-specific page boundary, you must know the page size supported by the machine on which your application is being run. You can obtain the CPU-specific page size at run time by calling the $GETSYI system service or the LIB$GETSYI run-time library routine, and use this value to calculate an aligned end-address value to pass in the **inadr** argument to the system services.

  Note that you should specify the **retadr** argument to determine the amount of usable memory the system mapped. The operating system maps a minimum of one page; however, your application may use only part of the page. The end-address specified in the **retadr** argument marks the upper limit of usable memory. (On Alpha systems, if your application specifies the **relpag** argument to the $CRMPSC system service, you *must* specify the **retadr** argument.)

For example, the VAX program in Example 6–4 maps the section file created in Section 6.3.1 into its existing virtual address space. The application defines a buffer, named `buffer`, that is 512 bytes in size, reflecting the VAX page size. The program defines the exact bounds of the section by passing the address of the first byte of the buffer as the start-address and the address of the last byte of the buffer as the end-address in the **inadr** argument.

Example 6–4 Mapping a Section into a Defined Area of Virtual Address Space

```
#include <ssdef.h>
#include <stdio.h>
#include <stsdef.h>
#include <descrip.h>
#include <dvidef.h>
#include <rms.h>
#include <secdef.h>

struct FAB fab;

char *filename = "maptest.dat";

char _align(page) buffer[512];

main( argc, argv )
int argc;
char *argv[];
{
     int    status = 0;
     long   flags = 0;
     long   inadr[2];
     long   retadr[2];
     int    fileChannel;

/********  create disk file to be mapped *************/

     fab = cc$rms_fab;
     fab.fab$l_fna = filename;
     fab.fab$b_fns = strlen( filename );
     fab.fab$l_fop = FAB$M_CIF | FAB$M_UFO;  /* must be UFO */

     status = sys$create( &fab );

     if( status & STS$M_SUCCESS )
        printf("Opened mapfile %s\n",filename);
     else
     {
        printf("Cannot open mapfile %s\n",filename);
        exit( status );
     }

     fileChannel = fab.fab$l_stv;

/**********  create and map the section  ****************/

     inadr[0] = &buffer[0];
     inadr[1] = &buffer[511];

     printf("inadr[0]=%u,inadr[1]=%u\n",inadr[0],inadr[1]);

     status = SYS$CRMPSC(inadr, /* inadr=address target for map */
                       &retadr, /* retadr= what was actually mapped */
                             0, /* acmode  */
                             0, /* flags */
                             0, /* gsdnam, only for global sections */
                             0, /* ident, only for global sections */
                             0, /* relpag, only for global sections */
                   fileChannel, /* returned by SYS$CREATE */
                             0, /* pagcnt = size of sect. file used */
                             0, /* vbn = first block of file used */
                             0, /* prot = default okay */
                             0 ); /* page fault cluster size */
```

(continued on next page)

Example 6–4 (Cont.) Mapping a Section into a Defined Area of Virtual Address Space

```
    if( status & STS$M_SUCCESS )
    {
          printf("Map succeeded\n");
          printf("retadr[0]=%u,retadr[1]=%u\n",retadr[0],retadr[1]);
    }
    else
    {
          printf("Map failed\n");
          exit( status );
    }
}
```

To get the program in Example 6–4 to run correctly on an Alpha system, you must make the following modifications:

- You must ensure that the start-address of the section specified in the **inadr** argument is aligned on an Alpha page boundary and the end-address specified is aligned with the end of an Alpha page.
- You must ensure that when a larger page on an Alpha system is mapped, neighboring data is not overwritten.

One way to accomplish these goals is to isolate the program section that contains the section data in its own image section by using the SOLITARY program section attribute.

In the example, the section, named `buffer`, appears in the program section named buffer. (Program section creation is different in various programming languages on each platform. Check compiler documentation to ensure that the section is placed in its own program section.), The following link operation illustrates how to set the solitary attribute of this program section:

```
$  LINK MAPTEST, SYS$INPUT/OPT
PSECT_ATTR=BUFFER,SOLITARY
Ctrl/Z
```

To specify an end-address for the section buffer that is aligned with the end of a CPU-specific page boundary, obtain the CPU-specific page size at run time, subtract 1 from the returned value, and use it to take the address of the last element of the array. Pass this value as the second longword in the **inadr** argument. (To find out how to obtain the page size at run time, see Section 6.4.) Note that you do not need to change the allocation of the buffer into which the section is mapped.

To ensure that your application will run on an Alpha system with any page size, specify the /BPAGE=16 qualifier to force the linker to align image sections on 64KB boundaries. Note that the total amount of memory mapped may be much larger than the total amount of usable memory. The amount of usable memory is determined by the value of the page count argument (**pagcnt**) or the size of the section file, whichever is smaller. To avoid using memory that is not within the bounds of the section, use the values returned in the **retadr** argument.

Example 6–5 shows the source changes required for Example 6–4 to get it to run on an Alpha system.

Example 6–5 Source Code Changes Required to Run Example 6–4 on an Alpha System

```
#include  <ssdef.h>
#include  <stdio.h>
#include  <stsdef.h>
#include  <string.h>
#include  <stdlib.h>
#include  <descrip.h>
#include  <dvidef.h>
#include  <rms.h>
#include  <secdef.h>
#include  <syidef.h> ➊

char buffer[512];  ➋
char *filename = "maptest.dat";
struct FAB fab;

long  cpu_pagesize; ➌

struct itm {                          /* item list */
    short int    buflen;  /* length of buffer in bytes */
    short int  item_code;  /* symbolic item code */
    long          bufadr;  /* address of return value buffer */
    long      retlenadr;  /* address of return value buffer length */
  } itmlst[2]; ➍

main( argc, argv )
int argc;
char *argv[];
{
     int    i;
     int    status = 0;
     long   flags = SEC$M_EXPREG;
     long   inadr[2];
     long   retadr[2];
     int    fileChannel;
     char  *mapped_section;

/********  create disk file to be mapped *************/

     fab = cc$rms_fab;
     fab.fab$l_fna = filename;
     fab.fab$b_fns = strlen( filename );
     fab.fab$l_fop = FAB$M_CIF | FAB$M_UFO;  /* must be UFO */

     status = sys$create( &fab );

     if( status & STS$M_SUCCESS )
        printf("%s opened\n",filename);
     else
     {
        exit( status );
     }

     fileChannel = fab.fab$l_stv;
```

(continued on next page)

Example 6–5 (Cont.) Source Code Changes Required to Run Example 6–4 on an Alpha System

```
/**********  obtain the page size at run time  ****************/

     itmlst[0].buflen =  4;
     itmlst[0].item_code = SYI$_PAGE_SIZE;
     itmlst[0].bufadr =  &cpu_pagesize;
     itmlst[0].retlenadr = &cpu_pagesize_len;
     itmlst[1].buflen = 0;
     itmlst[1].item_code = 0;

❺    status = sys$getsyiw( 0, 0, 0, &itmlst, 0, 0, 0 );

     if( status & STS$M_SUCCESS )
     {
          printf("getsyi succeeds, page size = %d\n",cpu_pagesize);
     }
     else
     {
          printf("getsyi fails\n");
          exit( status );
     }
/**********  create and map the section  ****************/

     inadr[0] = &buffer[0];
     inadr[1] = &buffer[cpu_pagesize - 1]; ❻

     printf("address of buffer = %u\n", inadr[0] );

     status = SYS$CRMPSC(&inadr, /* inadr=address target for map */
                        &retadr, /* retadr= what was actually mapped */
                              0, /* acmode  */
                              0, /* no flags to set  */
                              0, /* gsdnam, only for global sections */
                              0, /* ident, only for global sections */
                              0, /* relpag, only for global sections */
                    fileChannel, /* returned by SYS$CREATE */
                              0, /* pagcnt = size of sect. file used */
                              0, /* vbn = first block of file used */
                              0, /* prot = default okay */
                              0); /* page fault cluster size */

     if( status & STS$M_SUCCESS )
     {
          printf("section mapped\n");
          printf("start address returned =%u\n",retadr[0]);
     }
     else
     {
          printf("map failed\n");
          exit( status );
     }
}
```

The items in the following list correspond to the numbered items in Example 6–5:

❶ The header file SYIDEF.H contains definitions of OpenVMS item codes for the $GETSYI system service.

❷ The buffer is defined without using the __align(page) storage descriptor. Because the page size cannot be determined until run time on OpenVMS Alpha systems, the DEC C for OpenVMS Alpha compiler aligns the data on the largest Alpha page size (64 KB) when __align(page) is specified.

❸ This structure defines the item list used to obtain the page size at run time.

❹ This variable will hold the page-size value returned.

❺ This call to the $GETSYI system service obtains the page size at run time.

❻ The end-address of the buffer is specified by subtracting 1 from the page-size value returned.

6.3.4 Mapping from an Offset into a Section File

Your application may map a portion of a section file by specifying the address at which to start the mapping as an offset from the beginning of the section file. You specify this offset by supplying a value to the **relpag** argument of the $CRMPSC system service. The value of the **relpag** argument specifies the page number relative to the beginning of the file at which the mapping should begin.

To preserve compatibility, the $CRMPSC system service interprets the value of the **relpag** argument in 512-byte units on both VAX systems and Alpha systems. Note, however, that because the CPU-specific page size on Alpha systems is larger than 512 bytes, the address specified by the offset in the **relpag** argument probably does not fall on a CPU-specific page boundary. The $CRMPSC system service can map virtual memory in CPU-specific page increments only. Thus, on Alpha systems, the mapping of the section file will start at the beginning of the CPU-specific page that contains the offset address, *not* at the address specified by the offset.

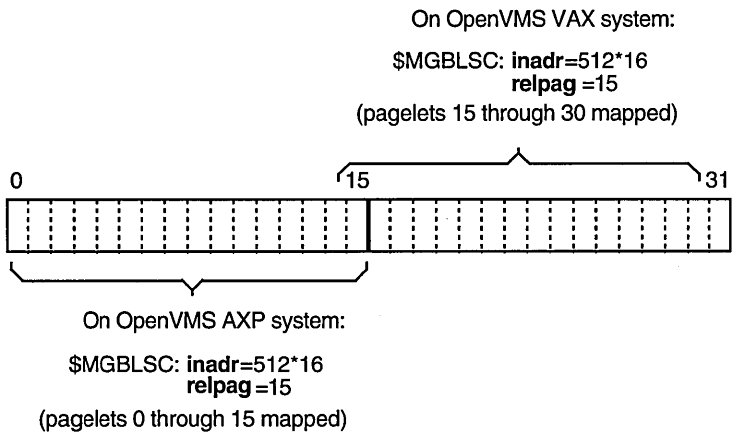
Note

Even though the routine starts mapping at the beginning of the CPU-specific page that contains the address specified by the offset, the start-address returned in the **retadr** argument is the address specified by the offset, *not* the address at which mapping actually starts.

If your application maps from an offset into a section file, you may need to enlarge the size of the address range specified in the **inadr** argument to accommodate the extra virtual memory space that gets mapped on Alpha systems. If the address range specified is too small, your application may not map the entire portion of the section file you desire, because the mapping begins at an earlier starting address in the section file.

For example, to map 16 blocks in a section file starting at block number 15 on a VAX system, you could specify an address range 16*512 bytes in size in the **inadr** argument and specify a value of 15 for the **relpag** argument. To accomplish this same mapping on an Alpha system, you must allow for the difference in page sizes. For example, on an Alpha system with an 8KB page size, the address specified by the **relpag** offset might fall 15 pagelets into a CPU-specific page, as shown in Figure 6–2. Because the $CRMPSC system service on an Alpha system begins the mapping of the section file at a CPU-specific page boundary, it would fail to map blocks 16 through 30. For the mapping to succeed, you would need to increase the size of the address range to accommodate the additional 15 pagelets mapped by the $CRMPSC system service (or the $MGBLSC system service) on an Alpha system. Otherwise, only one block of the portion of the section file you specified would be mapped.

Figure 6–2 Effect of Address Range on Mapping from an Offset



ZK–2499A–GE

When trying to calculate how much to enlarge the size of the address range specified in the **relpag** argument, the following formula may be helpful. The formula calculates the maximum number of CPU-specific pages needed to map a given number of pagelets.

$$\frac{(number\_of\_pagelets\_to\_map + (2 * pagelets\_per\_page) - 2)}{pagelets\_per\_page}$$

For example, this formula can be used to calculate how much to enlarge the address range specified in the previous scenario. In the following equation, the page size is assumed to be 8K, so *pagelets_per_page* equals 16:

```
16+((2x16)-2)/16=2.87...
```

Rounding the result down to the nearest whole number, the formula indicates that the address range specified in the **inadr** argument must encompass two CPU-specific pages.

6.4 Obtaining the Page Size at Run Time

To obtain the page size supported by an Alpha system, use the $GETSYI system service. Example 6–6 shows how to use this system service to obtain the page size at run time.

Example 6–6 Using the $GETSYI System Service to Obtain the CPU-Specific Page Size

```
#include  <ssdef.h>
#include  <stdio.h>
#include  <stsdef.h>
#include  <descrip.h>
```

(continued on next page)

Example 6–6 (Cont.) Using the $GETSYI System Service to Obtain the CPU-Specific Page Size

```
#include  <dvidef.h>
#include  <rms.h>
#include  <secdef.h>
#include  <syidef.h>  /* defines page size item code symbol */

struct itm {            /* define item list                       */
    short int    buflen;  /* length in bytes of return value buffer */
    short int  item_code;  /* item code                             */
    long         bufadr;  /* address of return value buffer       */
    long      retlenadr;  /* address of return value length buffer  */
  } itmlst[2];

long  cpu_pagesize;
long  cpu_pagesize_len;

main( argc, argv )
int argc;
char *argv[];
{
     int    status = 0;

     itmlst[0].buflen =  4;                /*  page size requires 4 bytes  */
     itmlst[0].item_code = SYI$_PAGE_SIZE; /*  page size item code         */
     itmlst[0].bufadr =  &cpu_pagesize;    /*  address of ret_val buffer   */
     itmlst[0].retlenadr = &cpu_pagesize_len; /* addr of length of ret_val */
     itmlst[1].buflen = 0;
     itmlst[1].item_code = 0;   /* Terminate item list with longword of 0  */

     status = sys$getsyiw( 0, 0, 0, &itmlst, 0, 0, 0 );

     if( status & STS$M_SUCCESS )
     {
          printf("getsyi succeeds, page size = %d\n",cpu_pagesize);
          exit( status );
     }
     else
     {
          printf("getsyi fails\n");
          exit( status );
     }
}
```

6.5 Locking Memory in the Working Set

The $LKWSET system service locks into the working set the range of pages identified in the **inadr** argument as an address range on both VAX and Alpha systems. The system service rounds the addresses to CPU-specific page boundaries if necessary.

However, because Alpha instructions cannot contain full virtual addresses, Alpha images must reference procedures and data indirectly through a pointer to a procedure descriptor. The procedure descriptor contains information about the procedure, including the actual code address. These pointers to procedure descriptors and data are collected into a new program section called a **linkage section**.

Recommendation

On Alpha systems, it is not sufficient to simply lock a section of code into memory to improve performance. You must also lock the associated linkage section into the working set.

To lock the linkage section in memory, determine the start- and end-addresses of the linkage section and pass these addresses as values in the **inadr** argument to a call to the $LKWSET system service.

7 Preserving the Integrity of Shared Data

This chapter describes synchronization mechanisms that ensure the integrity of shared data, such as the atomicity guaranteed by certain VAX instructions.

7.1 Overview

If your application uses multiple threads of execution and the threads share access to data, you may need to add explicit synchronization mechanisms to your application to protect the integrity of the shared data on Alpha systems. Without synchronization, an access to the data initiated by one application thread can potentially interfere with an access initiated simultaneously by a competing thread, leaving the data in an unpredictable state.

On VAX systems, the degree of synchronization required depends on the relationship of the different threads of execution, which can include the following:

- Multiple threads executing within a single process, such as a main thread interrupted by an asynchronous system trap (AST) thread.

  Note that the AST thread can either be initiated by the application or by the operating system. For example, the operating system uses an AST to write status to an I/O status block. The operating system also uses an AST to complete a buffered I/O read operation to a specified user buffer.
- Multiple threads separated into multiple processes executing on a single processor that access a global section.
- Multiple threads separated into multiple processes executing *concurrently* on multiple processors that access a global section.

On VAX systems, applications that take advantage of the parallel processing potential of a multiprocessor system have always had to provide explicit synchronization mechanisms such as locks, semaphores, and interlocked instructions to protect shared data. However, applications that use multiple threads on uniprocessor systems may not explicitly protect the shared data. Instead, these applications may depend on the implicit protection provided by features of the VAX architecture that guarantee synchronization between application threads executing on a VAX uniprocessor system (described in Section 7.1.1).

For example, applications that use a semaphore variable to synchronize access to a critical region of code by multiple threads depend on the semaphore being incremented atomically. On VAX systems, this is guaranteed by the VAX architecture. The Alpha architecture does not make the same synchronization guarantees. On Alpha systems, access to this semaphore or any data that can be accessed by multiple threads of execution must be explicitly synchronized. Section 7.1.2 describes features of the Alpha architecture you can use to provide equivalent protection.

7.1.1 VAX Architectural Features That Guarantee Atomicity

The following features of the VAX architecture provide synchronization among multiple threads of execution running on a uniprocessor system. (Note that the VAX architecture does not extend this guarantee of atomicity to multiprocessor systems.)

- **Instruction atomicity**—Many of the instructions defined by the VAX architecture are capable of performing a read-modify-write operation in a single, noninterruptable sequence (called an **atomic** operation) from the viewpoint of multiple application threads executing on a single processor. The Alpha architecture does not support such instructions. Operations that could be performed atomically on VAX systems require a sequence of instructions on Alpha systems, which can be interrupted, leaving the data in an unpredictable state.

  For example, the VAX Increment Long (INCL) instruction fetches the contents of a specified longword, increments its value, and stores the value back in the longword, performing the operations without interruption. On Alpha systems, each step must be explicitly performed by a separate instruction.

  To provide compatibility with VAX systems, the Alpha architecture defines a pair of instructions that you can use to ensure that a read/write operation is done atomically. Section 7.1.2 describes these instructions and how compilers on Alpha systems make this capability available to programs written in high-level languages.

  Note, however, that even on VAX systems, implicit dependence on the atomicity of VAX instructions is not recommended. Because of the optimizations they perform, compilers on VAX systems do not guarantee that they implement certain program statements, such as an increment operation (x = x + 1), using a VAX atomic instruction, even if such an instruction is available.

- **Memory access granularity**—The VAX architecture supports instructions that can manipulate byte- and word-sized data in a single, noninterruptable operation. (The VAX architecture supports instructions to manipulate data of other sizes as well.) The Alpha architecture supports instructions that manipulate longword- and quadword-sized data. Manipulation of byte- and word-sized data on Alpha systems requires multiple instructions: the longword or quadword that contains the byte or word must be fetched, the nontargeted bytes must be masked, the target byte or word manipulated, and then the entire longword or quadword must be stored. Because this sequence is interruptable, operations on byte and word data, which are atomic on VAX systems, are not atomic on Alpha systems.

  Note that this change in the granularity of memory access can also affect the definition of which data is shared. On VAX systems, a byte- or word-sized data item that is shared can be manipulated individually. On Alpha systems, the entire longword or quadword that contains the byte- or word-sized item must be manipulated. Thus, simply because of its proximity to an explicitly shared data item, neighboring data may become *unintentionally* shared.

  Compilers use the Alpha instructions described in Section 7.1.2 to ensure the integrity of byte- and word-sized data.

- **Read/write ordering**—On VAX uniprocessor and multiprocessor systems, sequential write operations and read operations appear to occur in the same order in which you specify them from the viewpoint of all types of external threads of execution. Alpha uniprocessor systems also guarantee that the order of read and write operations appears synchronized for multiple threads of execution running within a single process or within multiple processes running on a uniprocessor. However, write operations visible to threads executing concurrently on an Alpha multiprocessor system require explicit synchronization.

  To provide compatibility with VAX systems, the Alpha architecture supports an instruction with which you can ensure that read/write operations occur in the order specified, from the viewpoint of all the processors in the system. Section 7.1.2 provides more information about this instruction and about how high-level languages make this instruction available. Section 7.3 describes the feature of the Alpha architecture that provides this synchronization and how the compilers make it available to high-level language programs on Alpha systems.

7.1.2 Alpha Compatibility Features

To provide compatibility with the atomicity capabilities of the VAX architecture, the Alpha architecture defines two mechanisms:

- **Load-locked/Store-conditional instructions**—The Alpha instruction set includes a pair of instructions, named Load-locked (LD*x*L) and Store-conditional (ST*x*C), that provide for atomic load and store operations by setting and testing a lock bit. For complete information about these instructions, see the *Alpha Architecture Reference Manual*.

  Using the Load-locked/Store-conditional instructions, compilers can provide atomic access to byte- and word-sized data on Alpha systems. In addition, compilers may generate the Load-locked/Store-conditional instruction sequence when accessing byte- and word-sized data that is declared with the **volatile** attribute. (The Alpha architecture provides atomic load and store operations of longword- and quadword-sized data.)

- **Memory barriers**—The Alpha instruction set includes an instruction that can ensure that read/write operations, issued by multiple threads executing on separate processors in a multiprocessor system, appear to occur in the order specified. This instruction, named memory barrier (MB), guarantees that all subsequent load or store instructions will not access memory until after all previous load and store instructions have accessed memory from the viewpoint of multiple threads of execution.

7.2 Uncovering Atomicity Assumptions in Your Application

One way to uncover synchronization assumptions in your application is to identify data that is shared among multiple threads of execution and then examine each access to the data from each thread. When looking for shared data, remember to include unintentionally shared data as well as intentionally shared data. Unintentionally shared data is shared because of its proximity to data that is accessed by multiple threads of execution such as data written to by ASTs generated by the operating system as a result of system services such as $QIO, $ENQ, or $GETJPI.

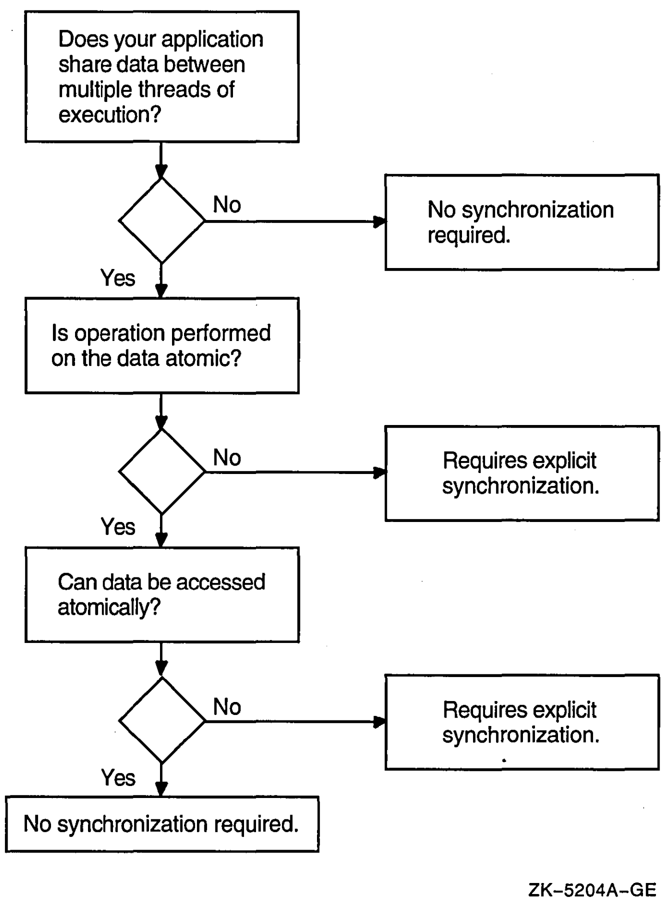
Because compilers on Alpha systems use quadword instructions by default in certain circumstances, all data items within a quadword of a shared data item may potentially become unintentionally shared. For example, compilers use quadword instructions to access a data item that is not aligned on natural boundaries. (Data is naturally aligned when its address is divisible by its size. For more information, see Chapter 8. Compilers align explicitly declared data on natural boundaries by default.)

When examining data access, determine if another thread could view the data in an intermediate state and, if such a view is possible, whether it is important to the application. In some cases, the exact value of the shared data may not be important; the application depends only on the relative value of the variable. In general, ask the following questions:

- Is the operation performed on the shared data atomic from the viewpoint of other threads of execution?
- Is it possible to perform an atomic operation to the data type involved?

Figure 7–1 shows this decision process.

Figure 7–1 Synchronization Decision Tree



7.2.1 Protecting Explicitly Shared Data

Example 7–1 is a simplified example of some possible atomicity assumptions in a VAX application. The program uses a variable, *flag*, through which an AST thread communicates with a main processing thread of execution. The main processing loop continues working until the counter variable reaches a predetermined value. The program queues an AST interruption that sets the flag to the maximum value, terminating the processing loop.

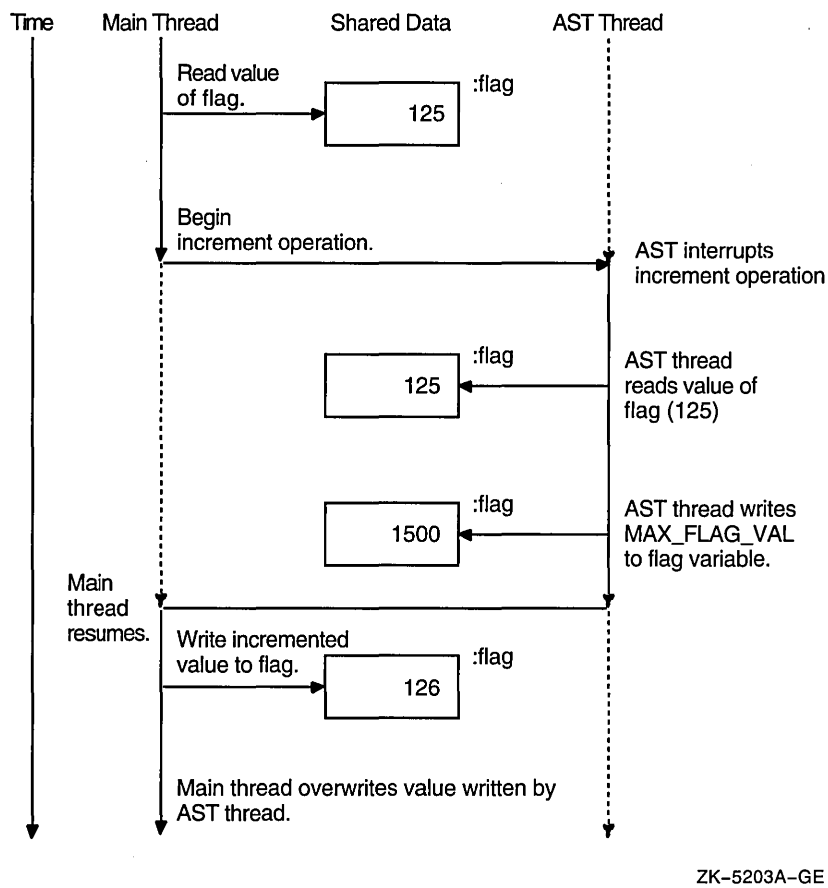
Example 7–1 Atomicity Assumptions in a Program with an AST Thread

```
#include <ssdef.h>
#include <descrip.h>

#define MAX_FLAG_VAL 1500

int    ast_rout();
long  time_val[2];
short int     flag;    /* accessed by main and AST threads */

main( )
{
     int      status = 0;
     static  $DESCRIPTOR(time_desc, "0 ::1");

     /*  changes ASCII time value to binary value  */

     status = SYS$BINTIM(&time_desc, &time_val);

     if ( status != SS$_NORMAL )
     {
        printf("bintim failure\n");
        exit( status );
     }

     /*  Set timer, queue ast */

     status = SYS$SETIMR( 0, &time_val, ast_rout, 0, 0 );

     if ( status != SS$_NORMAL )
     {
        printf("setimr failure\n");
        exit( status );
     }

     flag = 0;    /* loop until flag = MAX_FLAG_VAL */
     while( flag < MAX_FLAG_VAL )
     {
           printf("main thread processing (flag = %d)\n",flag);
           flag++;
     }
     printf("Done\n");
}

ast_rout()     /*  sets flag to maximum value to stop processing */
{
      flag = MAX_FLAG_VAL;
}
```

In Example 7–1, the variable named *flag* is explicitly shared between the main thread of execution and an AST thread. The program does not use any synchronization mechanism to protect the integrity of this variable; it implicitly depends on the atomicity of the increment operation.

On an Alpha system, this program may not always work as desired because the mainline thread of execution can be interrupted in the middle of the increment operation by the AST thread before the new value is stored back into memory, as shown in Figure 7–2. (This would be more likely to fail in a real application with dozens of AST threads.) In this scenario, the AST thread would interrupt the increment operation before it completes, setting the value of the variable to the maximum value. But once control returns to the main thread, the increment operation would complete, overwriting the value of the AST thread. When the loop test is performed, the value would not be at its maximum and the processing loop would continue.

Figure 7–2 Atomicity Assumptions in Example 7–1



Recommendations

To correct these atomicity dependencies, Digital recommends doing the following:

- Disable AST delivery, using the $SETAST system service, while the data is being accessed and enable it after access is completed.
- Explicitly protect the data by using a compiler mechanism. For example, DEC C for OpenVMS Alpha systems supports atomicity built-ins. In addition, you can use other mechanisms to synchronize access to this data, such as the $ENQ system service (for data accessed by multiple threads running on a multiprocessor system) or run-time library routines, such as LIB$BBCCI or LIB$BBSSI, and the interlocked queue routines.

For example, in Example 7–1, replace the increment operation, which is performed by the C increment operator (flag++) with the atomicity built-in supported by DEC C for OpenVMS Alpha systems (__ADD_ATOMIC_LONG(&flag,1,0)). See Example 7–2 for the complete example.

Note that the shared variable must be an aligned longword or aligned quadword to be protected by the atomicity built-ins.

- If you cannot change byte- or word-sized data to a longword or quadword, change the granularity the compiler uses when accessing the data item. Many compilers on Alpha systems allow you to specify the granularity they will use when accessing a particular data item or when processing an entire module. Note, however, that specifying byte and word granularity can have an adverse effect on the performance of your application.

Example 7–2 shows how these changes are implemented in the program presented in Example 7–1.

Example 7–2 Version of Example 7–1 with Synchronization Assumptions

```
#include <ssdef.h>
#include <descrip.h>
#include <builtins.h> ❶

#define MAX_FLAG_VAL  1500
int    ast_rout();
long  time_val[2];
int ❷       flag;    /* accessed by mainline and AST threads */

main( )
{
     int      status = 0;
     static  $DESCRIPTOR(time_desc, "0 ::1");

     /*  changes ASCII time value to binary value  */

     status = SYS$BINTIM(&time_desc, &time_val);

     if ( status != SS$_NORMAL )
     {
        printf("bintim failure\n");
        exit( status );
     }

     /*  Set timer, queue ast */

     status = SYS$SETIMR( 0, &time_val, ast_rout, 0, 0 );

     if ( status != SS$_NORMAL )
     {
        printf("setimr failure\n");
        exit( status );
     }
```

(continued on next page)

Example 7–2 (Cont.) Version of Example 7–1 with Synchronization Assumptions

```
     flag = 0;
     while( flag < MAX_FLAG_VAL )   /* perform work until flag set to zero */
     {
          printf("mainline thread processing (flag = %d)\n",flag);
          __ADD_ATOMIC_LONG(&flag,1,0); ❸
     }
     printf("Done\n");
}
ast_rout()     /*  sets flag to maximum value to stop processing */
{
      flag = MAX_FLAG_VAL;
}
```

The items in the following list correspond to the numbers in Example 7–2:

❶ To use the DEC C for OpenVMS Alpha systems atomicity built-ins, you must include the builtins.h header file.

❷ In this version, the variable *flag* is declared as a longword to allow atomic access (the atomicity built-ins require it).

❸ The increment operation is performed with an atomicity built-in function.

7.2.2 Protecting Unintentionally Shared Data

In Example 7–1, both threads clearly access the same variable. However, on an Alpha system, it is possible for an application to have atomicity concerns for variables that are inadvertently shared. In this scenario, two variables are physically adjacent to each other within the boundaries of a longword or quadword. On VAX systems, each variable can be manipulated individually. On an Alpha system, which supports atomic read and write operations of longword and quadword data only, the entire longword must be fetched before the target bytes can be modified. (For more information about this change in data-access granularity, see Chapter 8.)

To illustrate this problem, consider a modified version of the program in Example 7–1 in which the main thread and the AST thread each increment separate counter variables that are declared in a data structure, as in the following code:

```
struct {
    short int    flag;
    short int ast_flag;
    };
```

If both the main thread and the AST thread attempt to modify their individual target words simultaneously, the results would be unpredictable, depending on the timing of the two operations.

Recommendations

To remedy this synchronization problem, Digital suggests doing the following:

- Change the size of the shared variables to longwords or quadwords. Note, however, that because compilers on Alpha systems use quadword instructions in certain circumstances, you should use quadwords to ensure the integrity of the data. For example, if the data is not aligned on a natural boundary, the compilers use a quadword instruction to access the data.

  In data structures, you can also insert extra bytes between data items to force the elements of the structure onto natural quadword boundaries. The compilers align data on natural boundaries by default on Alpha systems.

  For example, to ensure that each flag variable in the data structure can be modified without interference from other threads of execution, change the declarations of the variables so that they are 64-bit quantities. Using DEC C, you could use the `double` data type, as in the following code:

  ```
  struct {
      double     flag;
      double  ast_flag;
      };
  ```

- Explicitly protect the data by using a compiler mechanism, such as the atomicity built-ins or the **volatile** attribute. In addition, you can synchronize access to data by multiple threads of execution running on a multiprocessor system by using the $ENQ system service or a run-time library routine, such as LIB$BBCCI or LIB$BBSSI, or by using interlocked queue operations.

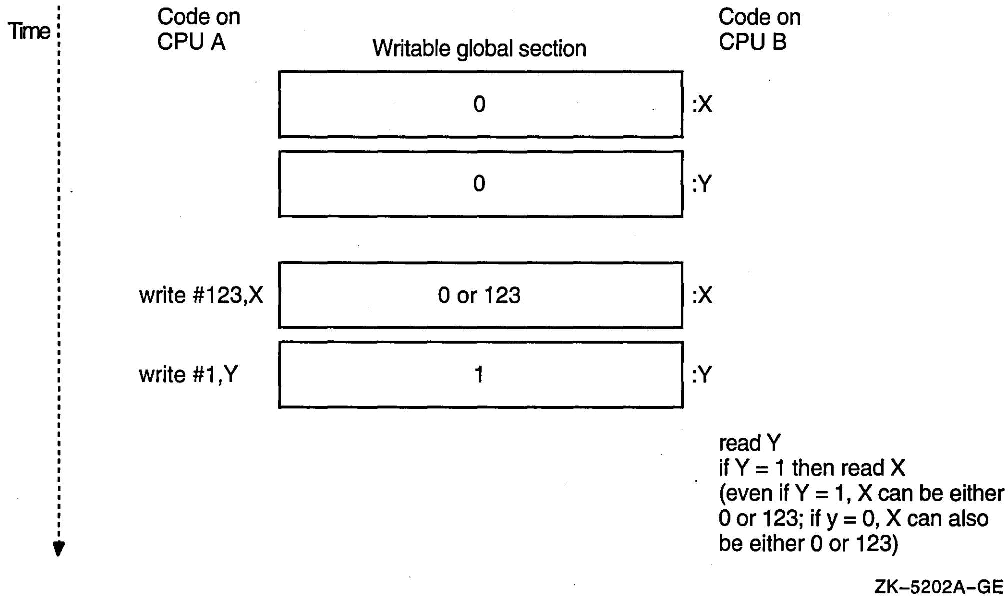
7.3 Synchronizing Read/Write Operations

VAX multiprocessing systems have traditionally been designed so that if one processor in a multiprocessing system writes multiple pieces of data, these pieces become visible to all other processors in the same order in which they were written. For example, if CPU A writes a data buffer (represented by X in Figure 7–3) and then writes a flag (represented by Y in Figure 7–3), CPU B can determine that the data buffer has changed by examining the value of the flag.

On Alpha systems, read and write operations to memory may be reordered to benefit overall memory subsystem performance. Processes that execute on a single processor can rely on write operations from that processor becoming readable in the order in which they are issued. However, multiprocessor applications cannot rely on the order in which write operations to memory become visible throughout the system. In other words, write operations performed by CPU A may become visible to CPU B in an order different from that in which they were written.

Figure 7–3 depicts this problem. CPU A requests a write operation to X, followed by a write operation to Y. CPU B requests a read operation from Y and, seeing the new value of Y, initiates a read operation of X. If the new value of X has not yet reached memory, CPU B receives the old value. As a result, any token-passing protocol relied on by procedures running on CPUs A and B is broken. CPU A could write data and set a flag bit, but CPU B may see the flag bit set *before* the data is actually written and erroneously use stale memory contents.

Figure 7–3 Order of Read and Write Operations on an Alpha System



Recommendations

Programs that run in parallel and that rely on read/write ordering require some redesigning to execute correctly on an Alpha system. One or more of the following techniques may be appropriate, depending on the application:

- Use the Alpha memory barrier instruction (MB) before and after all read and write instructions for which the completion order is crucial. For example, the DEC C for OpenVMS Alpha systems compiler supports the memory barrier instruction as a built-in function.
- Redesign the application to use the memory interlocks available in the PPL$ run-time library or the VAX interlocked instruction routines available in the LIB$ run-time library.
- Redesign the application to use the $ENQ and $DEQ system services to protect the data with a lock.

7.4 Ensuring Atomicity in Translated Images

The VEST command's /PRESERVE qualifier accepts keywords that allow translated VAX images to run on Alpha systems with the same guarantees of atomicity that are provided on VAX systems. Several /PRESERVE qualifier keywords provide different types of atomicity protection. Note that specifying these /PRESERVE qualifier keywords can have an adverse effect on the performance of your application. (For complete information about specifying the /PRESERVE qualifier, see *DECmigrate for OpenVMS AXP Systems Translating Images*.)

To ensure that an operation that can be performed atomically on a VAX system by a VAX instruction is performed atomically in a translated image, specify the INSTRUCTION_ATOMICITY keyword to the /PRESERVE qualifier.

To ensure that simultaneous updates to adjacent bytes within a longword or quadword can be accomplished without interfering with each other, specify the MEMORY_ATOMICITY keyword to the /PRESERVE qualifier.

To ensure that read/write operations appear to occur in the order you specify them, specify the READ_WRITE_ORDERING keyword to the /PRESERVE qualifier.

8
Checking the Portability of Application Data Declarations

This chapter describes how to check the data your application uses for dependencies on the VAX architecture. The chapter also describes the effect your choice of data type can have on the size and performance of your application on an Alpha system.

8.1 Overview

The data types supported by high-level programming languages, such as int in C or INTEGER*4 in FORTRAN, provide applications with a degree of data portability because they hide the machine-specific details of the underlying native data types. The languages map their data types to the native data types supported by the target platform. For this reason, you may be able to successfully recompile and run an application that runs on VAX systems on an Alpha system without modifying the data declarations it contains.

However, if your application contains any of the following assumptions about data types, you may need to modify your source code:

- **Assumptions about data-type mappings**—Your application may depend on the underlying VAX data type to which a high-level language maps. The Alpha architecture supports most of the VAX data types; however, there are some data types that are not supported. Your application may make assumptions about the size or bit format of a data type that may no longer be valid on an Alpha system. Section 8.2 provides more information about this topic.
- **Assumptions about data-type selection**—Your choice of data type may have different implications on an Alpha system. For example, on VAX systems, you may have chosen the smallest data type available to represent data items to conserve memory usage. On an Alpha system, this strategy may actually increase memory usage. Section 8.3 provides more information about this topic.

8.2 Checking for Dependence on a VAX Data Type

To provide data compatibility, the Alpha architecture has been designed to support many of the same native data types as the VAX architecture. Table 8–1 lists the native data types supported by both architectures. (See the *Alpha Architecture Reference Manual* for more information about the formats of the data types.)

Table 8–1 Comparison of VAX and Alpha Native Data Types

VAX Data Types	Alpha Data Types
byte	byte
word	word
longword	longword
quadword	quadword
octaword	–
F_floating	F_floating
D_floating (56-bit precision)	D_floating (53-bit precision)
G_floating	G_floating
H_floating	X_floating
–	S_floating (IEEE)
–	T_floating (IEEE)
Variable-length bit field	–
Absolute queue	Absolute longword queue
–	Absolute quadword queue
Self-relative queue	Self-relative longword queue
–	Self-relative quadword queue
Character string	–
Trailing numeric string	–
Leading separate numeric string	–
Packed decimal string	–

Recommendations

Unless your application depends on the format or size of the underlying native VAX data types, you may not have to modify your application because of changes to the data-type mappings. Wherever possible, the compilers on Alpha systems map their data types to the same native data types as they do on VAX systems. For those VAX data types that are not supported by the Alpha architecture, the compilers map their data types to the closest equivalent native Alpha data type. (For more information about how the compilers on Alpha systems map the data types they support to native Alpha data types, see Chapter 12 and compiler documentation.)

The following list provides guidelines that can be helpful for certain types of data declarations:

- **D_floating data**—Most compilers on Alpha systems map their double-precision floating-point data type to the VAX native G_floating data type by default because the Alpha architecture does not support the VAX D_floating data type. The OpenVMS VAX compilers map their double-precision floating-point data type to the D_floating data type. For example, VAX C maps the double data type to D_floating and DEC C for OpenVMS Alpha systems compiler maps the double data type to the G_floating data type.

  This change may not affect most applications. Note, however, that the value returned by the G_floating data type (significant to 15 digits after the decimal) is slightly less precise than the value returned by the D_floating data type (significant to 16 digits after the decimal).

The OpenVMS Run-Time Library supports a conversion routine (CVT$CONVERT_FLOAT) that can convert floating-point data from one format to another. For example, using this routine you can convert data in D_floating format to IEEE format and back again. Note also that the Alpha architecture supports the IEEE double-precision floating-point format (T_floating).

DEC C for OpenVMS Alpha systems issues a warning message when it encounters declarations that use the long float data type. On VAX systems, the long float data type is a synonym for double. On Alpha systems, the long float data type is obsolete, even when the DEC C compiler is used in VAX C mode.

- **Pointer data**—Check for assumptions that an address (pointer) data type is equivalent in size to an integer data type. On Alpha systems, an address is 64 bits.

  For example, in VAX C, some programs may make this assumption, as shown in Example 8–1.

Example 8–1 Assumptions About Data Types in VAX C Code

```
typedef struct {
       char     small;
       short   medium;
       long     large;
       } MYSTRUCT ;

main()
{
    int         a1;
    long        b1;
    MYSTRUCT    c1;

❶   a1 = &c1;
❷   b1 = &c1;

❸   a1->small = 1;
    b1->small = 2;
}
```

The items in the following list correspond to the numbered items in Example 8–1:

❶ The example assigns the address of the structure to the variable *a1*, declared as an int data type.

❷ The example assigns the address of the structure to the variable *b1*, declared as a long data type.

❸ The example accesses the first field in the structure by using the variables assigned to int and long data types.

To move this example to an Alpha system, you should change the declarations of *a1* and *b1* to be pointers to the data structure (MYSTRUCT), as in the following:

```
MYSTRUCT *a1,*b2;
```

8.3 Examining Assumptions About Data-Type Selection

Even though your application may recompile and run successfully on an Alpha system, your data-type selection may not take full advantage of the benefits of the Alpha architecture. In particular, data-type selection can impact the ultimate size of your application and its performance on an Alpha system.

8.3.1 Effect of Data-Type Selection on Code Size

On VAX systems, applications typically use the smallest size data type adequate for the data. For example, to represent a value between 32,768 and -32,767 in an application written in C, you might declare a variable of type short. On VAX systems, this practice conserves storage and, because the VAX architecture supports instructions that operate on all sizes of data types, does not affect efficiency.

On an Alpha system, byte- and word-sized data incurs more overhead than longword- or quadword-sized data because the Alpha architecture does not support instructions that manipulate these smaller data types. Each reference to a byte or word, which generates a single instruction on a VAX system, generates a sequence of instructions on an Alpha system, in which the longword containing the byte or word is fetched, manipulated so that only the target bytes are modified, and then stored. For frequently referenced data, these additional instructions can significantly add to the total size of your application on an Alpha system.

8.3.2 Effect of Data-Type Selection on Performance

Another aspect of data-type selection is data alignment. Alignment is an attribute of a data item that refers to its placement in memory. The mixture of byte-sized, word-sized, and larger data types, typically found in data-structure definitions and static data areas in applications on VAX systems, can lead to data that is not aligned on natural boundaries. (A data item is naturally aligned when its address is a multiple of its size in bytes.)

Accessing unaligned data incurs more overhead that accessing aligned data on both VAX and Alpha systems. However, VAX systems use microcode to minimize the performance impact of unaligned data. On Alpha systems, there is no hardware assistance. References to unaligned data trigger a fault, which must be handled by the operating system's unaligned fault handler. While the fault is being handled, the instruction pipeline must be stopped. Thus, the cost of an unaligned reference in performance is dramatically higher on Alpha systems.

The compilers on Alpha systems attempt to minimize the performance impact by generating a special unaligned reference instruction sequence when an unaligned reference is known at compile time. This prevents a run-time unaligned fault from occurring. Unaligned references that appear at run time must be handled as unaligned reference faults.

Recommendations

Given the potential impact of data-type selection on code size and performance, you might think you should change all byte- and word-sized data declarations to longwords to eliminate the extra instructions required for byte and word accesses and improve alignment. However, before making sweeping changes to your data declarations, consider the following factors:

- **Frequency of access/Number of replications**—If a byte- or word-sized data item is frequently referenced, changing it to a longword eliminates the extra instructions required at each reference and can reduce application size significantly. However, if the byte or word is not referenced frequently and is replicated a large number of times (for example, in a data structure instantiated many times), the change to a longword can add up to more than the cost of the additional instructions at each reference. The three bytes added when changing to a longword can significantly increase virtual memory usage if the data item is replicated thousands of times. Before changing a data declaration, consider how it is used and how much virtual memory (and thus physical memory) you want to spend for this performance improvement. Such trade-offs between size and performance are a frequent consideration during design.

- **Interoperability requirements**—If the data object is shared with a translated component or a native VAX component, you may be unable to make changes that would improve its layout because the other components depend on the binary layout of the data. Compilers (and the VEST utility) attempt to minimize the performance impact in this case by including the unaligned reference instruction sequence in the code they generate.

Taking these factors into consideration, use the following guidelines when examining data-type selections:

- For data that is frequently referenced but not frequently replicated, change byte- and word-sized fields to longwords, especially for performance-critical fields.
- For data that is *not* frequently referenced but that is frequently replicated, no change is recommended.
- For data that is both frequently referenced and frequently replicated, the decision must be made after carefully examining the code size versus performance impact of the change.
- For static data, always use a longword instead of a byte. It does incur three extra bytes of storage; however, a single reference requires three extra instructions, each of which is a longword.
- Use the capabilities of the compilers on Alpha systems to uncover data that is not aligned on natural boundaries. For example, many compilers on Alpha systems (for example, DEC C, Digital Fortran, but *not* DEC COBOL) support the /WARNING=ALIGNMENT qualifier, which checks for data that is not aligned on natural boundaries.
- Use the capabilities of the run-time analysis tools, Program Coverage and Analyzer (PCA) and the OpenVMS Debugger, to uncover at run time data that is not aligned on natural boundaries. For more information, see the *Guide to Performance and Coverage Analyzer for VMS Systems* and the *OpenVMS Debugger Manual.*
- Take advantage of the natural alignment provided by the compilers on Alpha systems, wherever interoperability concerns allow. On Alpha systems, compilers align data on natural boundaries by default, wherever possible. On VAX systems, compilers use byte alignment.

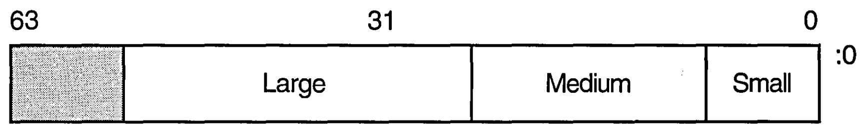
  Note that the compilers on Alpha systems support qualifiers and language pragmas that allow you to request they use the same byte alignment they use on VAX systems. For example, the DEC C for OpenVMS Alpha systems compiler supports the /NOMEMBER_ALIGNMENT qualifier and a corresponding pragma that allow you to control data alignment. For more information, see the DEC C compiler documentation.

The data structure defined in Example 8–1 shows these data-type selection concerns. The structure definition, called mystruct, is made up of byte-, word-, and longword-sized data, as follows:

```
struct{
      char    small;
      short   medium;
      long    large;
      } mystruct ;
```

When compiled using VAX C, the structure is laid out in memory as shown in Figure 8–1.

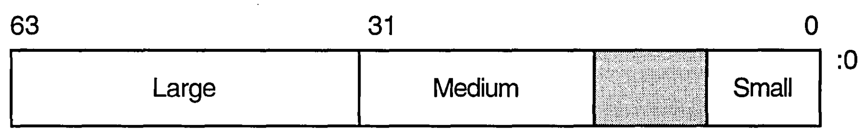
Figure 8–1 Alignment of mystruct Using VAX C



ZK–5209A–GE

When compiled using the DEC C for OpenVMS Alpha systems compiler, the structure is padded to achieve natural alignment, as shown in Figure 8–2. Note that by adding a byte of padding after the first field, small, both the following members of the structure are aligned.

Figure 8–2 Alignment of mystruct Using DEC C for OpenVMS Alpha Systems



ZK–5210A–GE

Note that the byte- and word-sized fields of the data structure still require multiple instruction sequences for access. If the fields small and medium are frequently referenced, and the entire structure is not frequently replicated, consider redefining the data structure to use longword data types. If, however, the fields are not frequently referenced or the data structure is frequently replicated, the cost of the byte or word references is a design trade-off the programmer must make.

9 Examining the Condition-Handling Code in Your Application

This chapter describes the effect of differences between the VAX architecture and the Alpha architecture on the condition-handling code in your application.

9.1 Overview

For the most part, the condition-handling code in your application will work correctly on an Alpha system, especially if your application uses the condition-handling facilities provided by the high-level language in which it is written, such as the END, ERR, and IOSTAT specifiers in FORTRAN. These language capabilities insulate applications from architecture-specific aspects of the underlying condition-handling facility.

However, there are certain differences between the Alpha condition-handling facility and the VAX condition-handling facility that may require you to modify your source code, including:

- Changes to the mechanism array format
- Changes to the condition codes returned by the system
- Changes to how other tasks related to condition handling in your application are accomplished, such as enabling exception signaling and specifying condition-handling routines dynamically at run time.

The following sections describe these changes in more detail and provide guidelines to help you decide if modifying your source code is necessary.

9.2 Establishing Dynamic Condition Handlers

The OpenVMS Alpha run-time libraries (RTLs) do not contain the routine LIB$ESTABLISH, which the OpenVMS VAX RTLs contain. Due to the nature of the OpenVMS Alpha calling standard, setting up condition handlers is done by compilers.

For those programs that need to dynamically establish condition handlers, some Alpha languages give special treatment for calls to LIB$ESTABLISH and generate the appropriate code without actually calling an RTL routine. The following languages support LIB$ESTABLISH semantics in a compatible fashion with the corresponding VAX language:

- DEC C and DEC C++

  Although DEC C and DEC C++ for OpenVMS Alpha systems treat LIB$ESTABLISH as a built-in function, the use of LIB$ESTABLISH is not recommended on OpenVMS VAX or OpenVMS Alpha systems. C and C++ programmers should call VAXC$ESTABLISH instead of LIB$ESTABLISH

(VAXC$ESTABLISH is a built-in function on DEC C and DEC C++ for OpenVMS Alpha systems).

- DEC Fortran

  DEC Fortran allows declarations to the LIB$ESTABLISH and LIB$REVERT intrinsic functions, and converts them to DEC Fortran RTL specific entry points.

- DEC Pascal

  DEC Pascal provides the built-in routines, ESTABLISH and REVERT, to use in place of LIB$ESTABLISH and LIB$REVERT. If you declare and try to use LIB$ESTABLISH, you will get a compile-time warning.

- MACRO–32

  The MACRO–32 compiler will attempt to call LIB$ESTABLISH if it is contained in the source code.

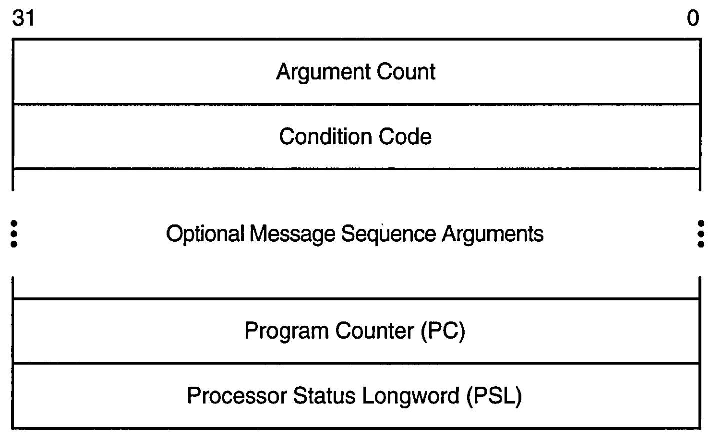
  If MACRO–32 programs establish dynamic handlers by storing a routine address at 0(FP), they will work correctly when compiled on an OpenVMS Alpha system. However, you cannot set the condition handler address from within a JSB (Jump to Subroutine) routine, only from within a CALL_ENTRY routine.

9.3 Examining Condition-Handling Routines for Dependencies

The calling sequence of user-written condition-handling routines remains the same on Alpha systems as it is on VAX systems. Condition-handling routines declare two arguments to access the data the system returns when it signals an exception condition. The system uses two arrays, the signal array and the mechanism array, to convey information that identifies which exception condition triggered the signal and to report on the state of the processor when the exception occurred.

The format of the signal array and the mechanism array is defined by the system and is documented in the Bookreader version of the *OpenVMS Programming Concepts Manual*. On Alpha systems, the data returned in the signal array and its format is the same as it is on VAX systems, as shown in Figure 9–1.

Figure 9–1 32-Bit Signal Array on VAX and Alpha Systems



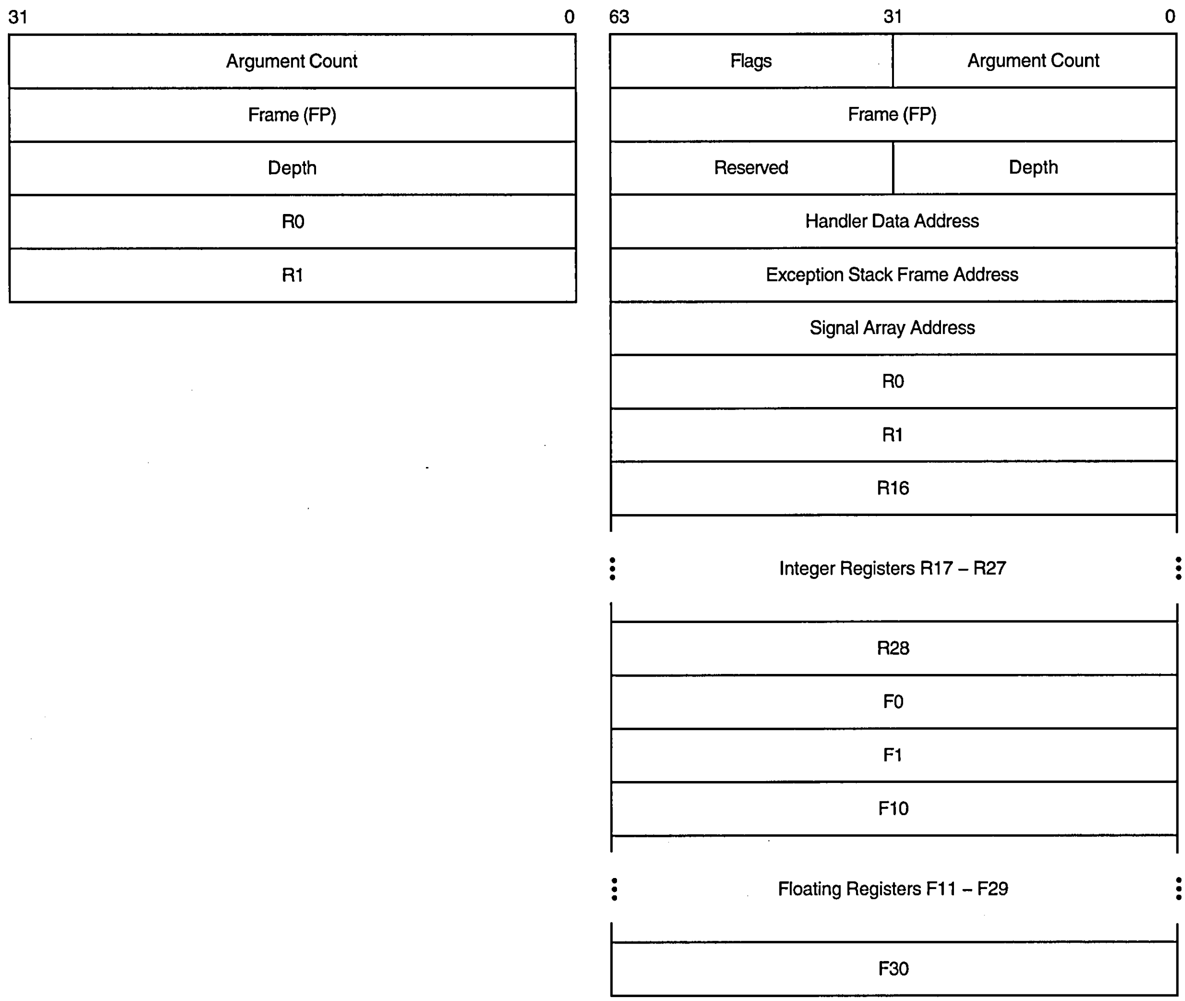
ZK–5208A–GE

The following table describes the arguments in the signal array:

Argument	Description
Argument Count	On Alpha and VAX systems, this argument contains a positive integer that indicates the number of longwords that follow in the array.
Condition Code	On Alpha and VAX systems, this argument is a 32-bit code that uniquely identifies a hardware or software exception condition. The format of the condition code, which remains unchanged on Alpha systems, is described in *OpenVMS Programming Interfaces: Calling a System Routine*. Note, however, that Alpha systems do not support every condition code returned on VAX systems and define condition codes that cannot be returned on a VAX system. Section 9.4 lists VAX condition codes that cannot be returned on Alpha systems.
Optional Message Sequence	These arguments provide additional information about the particular exception returned and vary for each exception. The Bookreader version of the *OpenVMS Programming Concepts Manual* describes these arguments for VAX exceptions.
Program Counter (PC)	The address of the next instruction to be executed when the exception occurred, if the exception is a trap; or the address of the instruction that caused the exception, if the exception is a fault. On Alpha systems, this argument contains the lower 32 bits of the PC (which is 64 bits long on Alpha systems).
Processor Status Longword (PSL)	A formatted 32-bit argument that describes the status of the processor when the exception occurred. On Alpha systems, this argument contains the lower 32 bits of the Alpha 64-bit processor status (PS) quadword.

On Alpha systems, the mechanism array returns much of the same data that it does on VAX systems; however, its format is different. The mechanism array returned on Alpha systems preserves the contents of a larger set of integer scratch registers as well as the floating-point scratch registers. In addition, because these registers are 64 bits long, the mechanism array is constructed of quadwords (64 bits) on Alpha systems, not longwords (32 bits) as it is on VAX systems. Figure 9–2 compares the format of the mechanism array on VAX and Alpha systems.

Figure 9–2 Mechanism Array on VAX and Alpha Systems



ZK–5207A–GE

The following table describes the arguments in the mechanism array:

Argument	Description
Argument Count	On VAX systems, this argument contains a positive integer that represents the number of longwords that follow in the array. On Alpha systems, this argument represents the number of *quadwords* in the mechanism array, not counting the argument count quadword (always 43 on Alpha systems).
Flags	On Alpha systems, this argument contains various flags to communicate additional information. For example, if bit 0 is set, it indicates that the process has already performed a floating-point operation and the floating-point registers in the array are valid (no equivalent in the mechanism array on VAX systems).

Argument	Description
Frame Pointer (FP)	On VAX and Alpha systems, this argument contains the address of the call frame on the stack that established the condition handler.
Depth	On VAX and Alpha systems, this argument contains an integer that represents the frame number of the procedure that established the condition-handling routine, relative to the frame that incurred the exception.
Reserved	Reserved.
Handler Data Address	On Alpha systems, this argument contains the address of the handler data quadword, when a handler is present (no equivalent in the mechanism array on VAX systems).
Exception Stack Frame Address	On Alpha systems, this argument contains the address of the exception stack frame (no equivalent in the mechanism array on VAX systems).
Signal Array Address	On Alpha systems, this argument contains the address of the signal array (no equivalent in the mechanism array on VAX systems).
Registers	On VAX and Alpha systems, the mechanism array includes the contents of scratch registers. On Alpha systems, this includes a much larger set of registers and also includes a corresponding set of floating-point registers.

Recommendations

Because the 32-bit signal array is the same on Alpha systems as it is on VAX systems, you may not need to modify the source code of your condition-handling routine. However, the changes to the mechanism array may require changes to your source code. In particular, check the following:

- Check the source code of your condition-handling routine for assumptions about the size of array elements or the ordering of array elements in the mechanism array.
- If the condition-handling routine in your application uses the depth argument to unwind a specific number of stack frames, you may need to modify your source code. Because of architectural changes, the depth argument returned on an Alpha system may be different from that returned on a VAX system. (The depth argument in the mechanism array indicates the number of frames between the procedure that established the handler, relative to the frame that incurred the exception.)

  Applications that unwind to the establisher frame by specifying the address of the depth argument to the SYS$UNWIND system service, or unwind to the caller of the establisher frame by using the default depth argument of the SYS$UNWIND system service, will continue to work correctly. Depths specified as negative numbers still indicate exception vectors (as on VAX systems).

Example 9–1 presents a condition-handling routine written in C.

Example 9–1 Condition-Handling Routine

```
#include  <ssdef.h>
#include  <chfdef.h>
   .
   .
   .
❶ int cond_handler( sigs, mechs )
   struct  chf$signal_array  *sigs;
   struct  chf$mech_array   *mechs;
{
     int status;

❷    status = LIB$MATCH_COND(sigs->chf$l_sig_name, /* returned code */
                               SS$_INTOVF);         /* test against  */

❸    if(status != 0)
     {
         /*  ...Condition matched.  Perform processing.  */
         return SS$_CONTINUE;
     }
     else
     {
         /*  ...Condition does not match. Resignal exception. */
         return SS$_RESIGNAL;
     }
}
```

The items in the following list correspond to the numbered items in Example 9–1:

❶ The routine defines two arguments, **sigs** and **mechs**, to access the data returned by the system in the signal array and the mechanism array. The routine declares the arguments using two predefined data structures, chf$signal_array and chf$mech_array, defined by the system in the CHFDEF.H header file.

❷ This condition-handling routine uses the LIB$MATCH_COND run-time library routine to compare the returned condition code with the condition code that identifies integer overflow (defined in SSDEF.H). The condition code is referenced as a field in the system-defined signal data structure (defined in CHFDEF.H).

❸ The LIB$MATCH_COND routine returns a nonzero result when a match is found. The condition-handling routine executes different code paths based on this result.

9.4 Identifying Exception Conditions

Application condition-handling routines identify which exception is being signaled by checking the condition code returned in the signal array. The following program fragment, taken from Example 9–1, shows how a condition-handling routine can accomplish this task by using the run-time library routine LIB$MATCH_COND:

```
status = LIB$MATCH_COND( sigs->chf$l_sig_name,  /* returned code */
                                  SS$_INTOVF);  /* test against  */
```

On Alpha systems, the format of the 32-bit condition code and its location in the signal array are the same as they are on VAX systems. However, the condition codes your condition-handling routine expects to receive on VAX systems may not be meaningful on Alpha systems. Because of architectural differences, some exception conditions that are returned on VAX systems are not supported on Alpha systems.

For software exceptions, Alpha systems support the same set supported by VAX systems, as documented in the online Help Message utility or in the OpenVMS system messages documentation. Hardware exceptions, however, are more architecture specific, especially the arithmetic exceptions. Only a subset of the hardware exceptions supported by VAX systems (documented in the Bookreader version of the *OpenVMS Programming Concepts Manual*) are also supported on Alpha systems. In addition, the Alpha architecture defines several additional exceptions that are not supported by the VAX architecture.

Table 9–1 lists the VAX hardware exceptions that are not supported on Alpha systems and the Alpha hardware exceptions that are not supported on VAX systems. If the condition-handling routine in your application tests for any of these VAX-specific exceptions, you may need to add the code to test for the equivalent Alpha exceptions. (Section 9.4.1 provides more information about testing for arithmetic exceptions on Alpha systems.)

Note

A translated VAX image run on an Alpha system can still return these VAX exceptions.

Table 9–1 Architecture-Specific Hardware Exceptions

Exception Condition Code	Comment
Exceptions Specific to Alpha Systems	
SS$_HPARITH–High-performance arithmetic exception	Replaces VAX arithmetic exceptions (see Section 9.4.1)
SS$_ALIGN–Data alignment trap	No equivalent on VAX systems
Exceptions Specific to VAX Systems	
SS$_ARTRES–Reserved arithmetic trap	No equivalent on Alpha systems
SS$_COMPAT–Compatibility fault	No equivalent on Alpha systems
SS$_DECOVF–Decimal overflow[1]	Replaced by SS$_HPARITH (see Section 9.4.1)
SS$_FLTDIV–Float divide-by-zero (trap)[1]	Replaced by SS$_HPARITH (see Section 9.4.1)
SS$_FLTDIV_F–Float divide-by-zero (fault)	Replaced by SS$_HPARITH (see Section 9.4.1)
SS$_FLTOVF–Float overflow (trap)[1]	Replaced by SS$_HPARITH (see Section 9.4.1)
SS$_FLTOVF_F–Float overflow (fault)	Replaced by SS$_HPARITH (see Section 9.4.1)
SS$_FLTUND–Float underflow (trap)[1]	Replaced by SS$_HPARITH (see Section 9.4.1)

[1]May be generated by software on Alpha systems

(continued on next page)

Table 9–1 (Cont.) Architecture-Specific Hardware Exceptions

Exception Condition Code	Comment
Exceptions Specific to VAX Systems	
SS$_FLTUND_F–Float underflow (fault)	Replaced by SS$_HPARITH (see Section 9.4.1)
SS$_INTDIV–Integer divide-by-zero[1]	Replaced by SS$_HPARITH (see Section 9.4.1)
SS$_INTOVF–Integer overflow[1]	Replaced by SS$_HPARITH (see Section 9.4.1)
SS$_TBIT–Trace pending	No equivalent on Alpha systems
SS$_OPCCUS–Opcode reserved to customer	No equivalent on Alpha systems
SS$_RADMOD–Reserved addressing mode	No equivalent on Alpha systems
SS$_SUBRNG–INDEX subscript range check	No equivalent on Alpha systems

[1]May be generated by software on Alpha systems

9.4.1 Testing for Arithmetic Exceptions on Alpha Systems

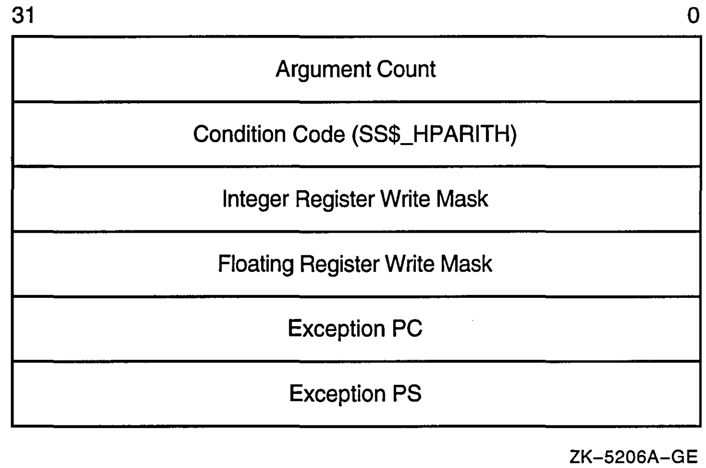
On a VAX system, the architecture ensures that arithmetic exceptions are reported synchronously; that is, a VAX arithmetic instruction that causes an exception (such as an overflow) enters any exception handlers immediately and no subsequent instructions are executed. The program counter (PC) reported to the exception handler is that of the failing arithmetic instruction. This allows application programs, for example, to resume the main sequence, with the failing operation being emulated or replaced by some equivalent or alternate set of operations.

On Alpha systems, arithmetic exceptions are reported asynchronously; that is, implementations of the architecture can allow a number of instructions (including branches and jumps) to execute beyond that which caused the exception. These instructions may overwrite the original operands used by the failing instruction, thus causing information integral to interpreting or rectifying the exception to be lost. The PC reported to the exception handler is not that of the failing instruction, but rather is that of some subsequent instruction. When the exception is reported to an application's exception handler, it may be impossible for the handler to fix up the input data and restart the instruction.

Because of this fundamental difference in arithmetic exception reporting, Alpha systems define a single condition code, SS$_HPARITH, to indicate all of the arithmetic exceptions. Thus, if your application contains a condition-handling routine that performs processing when an integer overflow exception occurs, on VAX systems it expects to receive the SS$_INTOVR condition code. On Alpha systems, this exception is indicated by the condition code SS$_HPARITH. In this way, condition-handling routines in applications cannot mistake an Alpha arithmetic exception with the corresponding VAX exception. This is important because the processing performed by the applications may be architecture specific.

Figure 9–3 shows the format of the SS$_HPARITH exception signal array.

Figure 9–3 SS$_HPARITH Exception Signal Array



This signal array contains three arguments that are specific to the SS$_HPARITH exception: the **integer register write mask**, the **floating register write mask**, and the **exception summary** arguments. The integer and floating register mask arguments indicate the registers that were targets of instructions that set bits in the exception summary argument. Each bit in the mask represents a register. The exception summary argument indicates the type of exception (or exceptions) that is being signaled by setting flags in the first seven bits. Table 9–2 lists the meaning of each of these bits when set.

Table 9–2 Exception Summary Argument Fields

Bit	Meaning
0	Software completion.
1	Invalid floating arithmetic, conversion, or comparison operation.
2	Invalid attempt to perform a floating divide operation with a divisor of zero. Note that integer divide-by-zero is not reported.
3	Floating arithmetic or conversion operation overflowed the destination exponent.
4	Floating arithmetic or conversion operation underflowed the destination exponent.
5	Floating arithmetic or conversion operation gave a result that differed from the mathematically exact result.
6	Integer arithmetic or conversion operation from floating point to integer overflowed the destination precision.

Recommendations

The following recommendations provide guidelines for determining if a condition-handling routine that performs processing in response to an arithmetic exception needs modification to run on an Alpha system:

- If the condition-handling routine in your application only counts the number of arithmetic exceptions that occurred, or aborts when an arithmetic exception occurs, it does not matter that the exception is delivered asynchronously on Alpha systems. These condition-handling routines require only the addition of a test for the SS$_HPARITH condition code.

- If your application attempts to restart the operation that caused the exception, you must either rewrite your code or use a compiler qualifier that ensures the exact reporting of arithmetic exceptions. (See Chapter 12 for more information about these compiler qualifiers.) Note, however, that specifying these instructions can affect performance adversely.
- To guarantee precise reporting of arithmetic exceptions in translated images, specify the /PRESERVE=FLOAT_EXCEPTIONS qualifier on the VEST command line when translating the image. When this qualifier is used, the VEST utility generates code that allows an exception to be reported after each instruction that could result in a floating-point fault. This qualifier adversely affects the performance of the translated image. For more information about using the VEST command, see *DECmigrate for OpenVMS AXP Systems Translating Images*.

Note

A translated VAX image running on an Alpha system can return VAX exception conditions, including arithmetic exception conditions.

9.4.2 Testing for Data-Alignment Traps

On an Alpha system, a data-alignment trap is generated when an attempt is made to load or store a longword or quadword to or from a register using an address that does not have the natural alignment of the particular data reference, without using an Alpha instruction that takes an unaligned address as an operand (LDQ_U). (For more information about data alignment, see Chapter 8.)

Compilers on Alpha systems typically avoid triggering alignment faults by:

- Aligning static data on natural boundaries by default. (This default behavior can be overridden by using a compiler qualifier.)
- Generating special inline code sequences for data that is known to be misaligned at compile time.

Note, however, that compilers cannot align dynamically defined data. Thus, alignment faults may be triggered.

An alignment exception is identified by the condition code SS$_ALIGN. Figure 9–4 shows the elements of the signal array returned by the SS$_ALIGN exception.

Figure 9–4 SS$_ALIGN Exception Signal Array

31   0
Argument Count
Condition Code (SS$_ALIGN)
Virtual Address
Register Number
Exception PC
Exception PS

ZK-5205A-GE

This signal array contains two arguments specific to the SS$_ALIGN exception: the **virtual address** argument and the **register number** argument. The virtual address argument contains the address of the unaligned data being accessed. The register number argument identifies the target register of the operation.

Recommendation

- Use this exception to detect alignment exceptions during the development of your application. In this phase, you have the opportunity to fix the data alignment before it can impact performance for a user of your application. Once this exception is reported, your application has already experienced the performance impact.

9.5 Performing Other Tasks Associated with Condition Handling

In addition to condition-handling routines, applications that include condition handling must perform other tasks, such as identifying their condition-handling routine to the system. The run-time library provides a set of routines that allow applications to perform these tasks. For example, applications can call the run-time library routine LIB$ESTABLISH to identify (or establish) the condition-handling routine they want executed when an exception is signaled.

Because of differences between the VAX architecture and the Alpha architecture and between the calling standards for both architectures, the way in which many of these tasks are accomplished is not the same. Table 9–3 lists the run-time library condition-handling support routines available on VAX systems and indicates which are supported on Alpha systems.

Table 9–3 Run-Time Library Condition-Handling Support Routines

Routine	Support on Alpha Systems
Arithmetic Exception Support Routines	
LIB$DEC_OVER–Enable or disable signaling of decimal overflow	Not supported
LIB$FIXUP_FLT–Change floating-point reserved operand to a specified value	Not supported
LIB$FLT_UNDER–Enable or disable signaling of floating-point underflow	Not supported
LIB$INT_OVER–Enable or disable signaling of integer overflow	Not supported
General Condition-Handling Support Routines	
LIB$DECODE_FAULT–Analyze instruction context for fault	Not supported
LIB$ESTABLISH–Establish a condition handler	Not supported by RTL but supported by compilers to provide compatibility
LIB$MATCH_COND–Match condition value	Supported
LIB$REVERT–Delete a condition handler	Not supported by RTL but supported by compilers to provide compatibility
LIB$SIG_TO_STOP–Convert a signaled condition to a condition that cannot be continued	Supported
LIB$SIG_TO_RET–Convert a signal to a return status	Supported
LIB$SIM_TRAP–Simulate a floating-point trap	Not supported
LIB$SIGNAL–Signal an exception condition	Supported
LIB$STOP–Stop execution by using signaling	Supported

Recommendations

The following list provides specific guidelines for applications that use run-time library routines:

- If your application enables the signaling of exceptions by calling one of the run-time library routines that enable exception reporting, you will need to change your source code. These routines are not supported on Alpha systems. Note, however, that certain types of arithmetic exceptions are always enabled on Alpha systems. The following types of arithmetic exceptions are always enabled:
  - Floating-point invalid operation
  - Floating-point division by zero
  - Floating-point overflow

  Those exceptions that are not enabled by default must be enabled at compile time.

- If your application specifies a condition-handling routine by calling the run-time library routine LIB$ESTABLISH, you may not have to change your source code. Most compilers on Alpha systems, to preserve compatibility, accept calls to the LIB$ESTABLISH routine. The compilers create a variable on the stack to point at the "current" condition handler. LIB$ESTABLISH sets this variable; LIB$REVERT clears it. The statically established handler for these languages reads the value of this variable to determine which routine to call. For information on specific languages, see Chapter 12.

The FORTRAN program in Example 9–2 uses the RTL routine LIB$ESTABLISH to specify a condition-handling routine that tests for integer overflow by specifying the condition code SS$_INTOVF. On VAX systems, you must compile the program with the /CHECK=OVERFLOW qualifier to enable integer overflow detection.

To get this program to run on an Alpha system, you must change the condition code from SS$_INTOVF to SS$_HPARITH. (You can determine the type of overflow by examining the exception summary argument in the signal array. For more information, see the compiler documentation.) As on VAX systems, you must specify the /CHECK=OVERFLOW qualifier on the compile command line to enable overflow detection. The call to the LIB$ESTABLISH routine does not have to be removed because DEC Fortran accepts this routine as an intrinsic function.

Example 9–2 Sample Condition-Handling Program

```
C       This program types a maximum value of integers
C       Compile with /CHECK=OVERFLOW and the /EXTEND_SOURCE qualifiers

        INTEGER*4 int4
        EXTERNAL HANDLER
        CALL LIB$ESTABLISH (HANDLER)  ❶

        int4=2147483645
        WRITE (6,*) ' Beginning DO LOOP, adding 1 to ', int4
        DO I=1,10
          int4=int4+1
          WRITE (6,*) ' INT*4 NUMBER IS  ', int4
        END DO
        WRITE (6,*) ' The end ...'
        END

C       This is the condition-handling routine

        INTEGER*4 FUNCTION HANDLER (SIGARGS, MECHARGS)
        INTEGER*4 SIGARGS(*),MECHARGS(*)
        INCLUDE '($FORDEF)'
        INCLUDE '($SSDEF)'
        INTEGER INDEX
        INTEGER LIB$MATCH_COND

        INDEX = LIB$MATCH_COND (SIGARGS(2), SS$_INTOVF) ❷
        IF (INDEX .EQ. 0 ) THEN
           HANDLER = SS$_RESIGNAL
        ELSE IF (INDEX .GT. 0) THEN
           WRITE (6,*) 'Arithmetic exception detected...'
           CALL LIB$STOP(SIGARGS(1))
        END IF
        END
```

The items in the following list correspond to the numbered items in Example 9–2:

1. The example calls LIB$ESTABLISH to specify the condition-handling routine.
2. On an Alpha system, you must change the condition code SS$_INTOVF to SS$_HPARITH. The handler routine calls the LIB$STOP routine to terminate execution of the program.

The following example shows how to compile, link, and run the program in Example 9–2.

```
$ FORTRAN/EXTEND_SOURCE/CHECK=OVERFLOW  HANDLER_EX.FOR
$ LINK  HANDLER_EX
$ RUN   HANDLER_EX
 Beginning DO LOOP, adding 1 to 2147483645
 INT*4 NUMBER IS    2147483646
 INT*4 NUMBER IS    2147483647
Arithmetic exception detected...
%TRACE-F-TRACEBACK, symbolic stack dump follows
 Image Name   Module Name     Routine Name    Line Number  rel PC      abs PC
 INT_OVR_HAND INT_OVR_HANDLER HANDLER                1637 00000238    00020238
 DEC$FORRTL                                             0 000651E4    001991E4
----- above condition handler called with exception 00000504:
%SYSTEM-F-HPARITH, high performance arithmetic trap, Imask=00000001, Fmask=00000
000, summary=40, PC=000200E0, PS=0000001B
-SYSTEM-F-INTOVF, arithmetic trap, integer overflow at PC=000200E0, PS=0000001B
----- end of exception message
                                                        0 84FE9FFC    84FE9FFC
 INT_OVR_HAND INT_OVR_HANDLER INT_OVR_HANDLER          15 000000E0    000200E0
                                                        0 84EFD918    84EFD918
                                                        0 7FF23EE0    7FF23EE0
```

10 Translating Applications

This chapter describes resources used to translate a VAX application to run on an Alpha system.

10.1 DECmigrate for OpenVMS Alpha

DECmigrate for OpenVMS Alpha is used to translate images for which the source code is not available. The VAX Environment Software Translator (VEST) component of DECmigrate translates the VAX binary image file into a native Alpha image. The translated image runs under the Translated Image Environment (TIE) on an Alpha computer. (TIE is a shareable image that is included with the OpenVMS Alpha operating system.) Translation does not involve running an OpenVMS VAX image under emulation or interpretation (with certain limited exceptions). Instead, the new OpenVMS Alpha image contains Alpha instructions that perform operations identical to those performed by the instructions in the original OpenVMS VAX image.

A translated image generally runs as fast on an Alpha computer as the original image runs on a VAX computer. However, a translated image does not benefit from the optimizing compilers that take full advantage of the Alpha architecture. Therefore, a translated image typically runs about 25 to 40 percent as fast as a native OpenVMS Alpha image. The primary causes for this reduced performance are unaligned data and the extensive use of complex VAX instructions.

DECmigrate translation support is limited to the language features, system services, and run-time library entry points that existed on OpenVMS VAX Version 5.5-2. This limitation and a method for overcoming it (in case your application uses features introduced after the OpenVMS VAX Version 5.5-2 release) are described in the *OpenVMS Version 6.2 Release Notes*.

A second function of DECmigrate is to analyze images to identify specific incompatibilities for an Alpha computer. Depending on the type of incompatibility, you can choose to specify a compiler qualifier that will compensate for the problem or make changes to the code.

For more information on image translation and VEST, see *DECmigrate for OpenVMS AXP Systems Translating Images*.

10.2 DECmigrate: Translated Image Support

Alpha

DECmigrate Version 1.1A runs on Alpha systems running OpenVMS Version 6.1 or later. The images it translates require this version or a later version to execute. Translated images are generally forward compatible but not backward compatible; that is, images translated with DECmigrate Version 1.1A can run only on Alpha systems running OpenVMS Version 6.1 or later while images translated with DECmigrate Version 1.0 can run on OpenVMS Alpha Version 1.0 and later. Table 10–1 correlates the versions of OpenVMS Alpha systems with the different versions of DECmigrate that support them.♦

Table 10–1 Support for Translated Images on OpenVMS Alpha Versions

DECmigrate Version Used to Translate Images	OpenVMS Alpha Support for Translated Images		
	Version 1.0	Version 1.5	Version 6.1 and later
Version 1.0	Yes	Yes	Yes
Version 1.1	No	Yes	Yes
Version 1.1A	No	No	Yes

10.3 Translated Image Environment (TIE)

Image translation is one means of migrating all or part of a VAX application to OpenVMS Alpha. The DECmigrate for OpenVMS AXP VAX Environment Software Translator utility (VEST) creates a translated image by converting a VAX executable or shareable image into a functionally equivalent Alpha image. VEST is a component of the optional layered product DECmigrate for OpenVMS AXP.

When a translated image runs on OpenVMS Alpha, the Translated Image Environment (TIE) provides the VAX environment required for the image to execute properly. The TIE consists of the shareable images TIE$SHARE and TIE$EMULAT_TV, which perform VAX complex instructions. For information on the role of image translation in a migration strategy, see the manuals *Migrating to an OpenVMS AXP System: Planning for Migration* and *DECmigrate for OpenVMS AXP Systems Translating Images*.

The following subsections discuss these topics:

- Interoperability between native and translated images
- Running translated images
- TIE statistics and feedback

Interoperability Between Native and Translated Images

The TIE works together with other components of OpenVMS Alpha to enable native and translated images to call one another. If you are developing applications or run-time libraries that rely on interoperability, you need to follow certain procedures when compiling, linking, or translating. See the first restriction described in Section 10.3.1.4. Table 10–2 provides pointers to documentation that describes the procedures.

Table 10–2 Interoperability Documentation

Goal	Reference
Ensuring interoperability between native and translated images	*Migrating to an OpenVMS AXP System: Recompiling and Relinking Applications* *DECmigrate for OpenVMS AXP Systems Translating Images*
Coordinating native and translated run-time libraries	*DECmigrate for OpenVMS AXP Systems Translating Images*

Running Translated Images

Use the DCL RUN command to run a translated image. For example:

```
$ RUN FOO_TV.EXE
```

Note that the translated image does not run correctly unless OpenVMS Alpha includes the appropriate translated shareable images and run-time libraries. When you translate an image, VEST requires the image information files (IIFs—file type .IIF) corresponding to whichever images and libraries the input image refers to. These .IIF files enable VEST to create a translated image that correctly refers to the translated versions of the shareable images and libraries. An image information file used at image translation must exactly correspond to the version of the translated shareable image or run-time library available on OpenVMS Alpha.

OpenVMS Alpha includes a set of translated run-time libraries and a matching set of image information files, which are listed in Section 10.4. Check these lists to determine if they include the libraries or shareable images referred to by images you want to translate and run. If OpenVMS Alpha does not include the required shared images or libraries, refer to *DECmigrate for OpenVMS AXP Systems Translating Images*. This manual describes how to create and use image information files.

When a translated library has been replaced by a native version of the library, you need to define accordingly any logical names that point to it—that is, you need to redefine *image_TV* to *image*.

TIE Statistics and Feedback

In addition to the TIE's run-time support function, TIE statistics and feedback can help to improve translated image performance:

- The TIE can display statistics about the run-time execution of translated images. These statistics describe the image's use of TIE resources and the interactions between images.
- The TIE can record information about VAX entry points discovered while interpreting VAX code. When you retranslate the image, VEST uses the information to find and translate more VAX code.

DECmigrate for OpenVMS AXP Systems Translating Images describes these features in detail and explains how to define the logical names that enable and disable their use.

10.3.1 Problems and Restrictions

This section describes known problems and restrictions with the TIE.

10.3.1.1 Condition Handler Restriction

There is a permanent restriction on the type of condition handler that can be established for both native and translated images. A native routine cannot establish a translated condition handler, nor can a translated routine establish a native condition handler. If a native or translated image violates this restriction, the run-time results are unpredictable.

10.3.1.2 Exception Handler Restrictions

The following exception handler restrictions are permanent:

- Translated images with exception handlers that depend on receiving the correct program status longword (PSL) might not function properly. When exceptions are reported, the Alpha program status (PS) is reported in the signal array instead because there is no VAX PSL.
- Translated images with exception handlers that depend on modifying the PSL in the signal array do not function properly. The modified PSL is not propagated back to the faulting code.

10.3.1.3 Floating-Point Restrictions

The following floating-point restrictions are permanent:

- In some cases, floating-point instructions operating on the same data generate a trap on an Alpha system but not on a VAX system. Specifically, VAX floating-point instructions on OpenVMS Alpha generate traps for the "dirty zeros" that VAX hardware can handle correctly. "Dirty zeros" are floating-point values that are alternate encodings for zero. To retain compatibility with translated code that performs operations using dirty zeros, the TIE includes a condition handler that corrects the dirty zeros and retries the floating-point operation. However, the handler succeeds only if the qualifier /PRESERVE=FLOAT_EXCEPTIONS was used when the image was translated.

  Images that were not translated with /PRESERVE=FLOAT_EXCEPTIONS and that perform an operation on a dirty zero incur an HPARITH exception with a summary status that has bit 1 set. If your translated application incurs one of these exceptions, retranslate with /PRESERVE=FLOAT_EXCEPTIONS. VAX dirty zeros commonly result from not initializing floating data to 0. In this case, changes to source code may be necessary to port to OpenVMS Alpha an application that uses dirty zeros.

- Alpha D53 floating point (D_floating point as a 53-bit fraction instead of a 56-bit fraction) is VAX D_floating converted to G_floating representation. This conversion leads to the following problem. Consider the VAX instruction sequence:

  ```
  MOVD    (SP),R2
  MOVD    R2,-(SP)
  ```

  VEST translates these VAX instructions into Alpha code like the following:

  ```
  LDD     F2,0(R14)       ! Pickup D float
  CVTDG   F2,F2           ! Convert to Canonical G Form with rounding
  CVTGD   F2,F17          ! Convert back to D Form for storing
  STD     F17,-8(R14)     ! Store the result
  ```

At run time, the VEST-generated code uses rounding to obtain the most accurate G_floating value when converting the D56 floating point to G canonical form. In some cases, the conversion to G canonical form may round up the D_floating value to create an exponent that cannot be represented in D_floating. When this happens, the CVTGD operation incurs an HPARITH trap with floating overflow as the summary reason.

If a translated image incurs this problem at run time, it needs to be retranslated with the VEST qualifier /FLOAT=D56_FLOAT to execute properly.

10.3.1.4 Interoperability Restrictions

Note the following interoperability restrictions:

- A native routine that either calls or is called by a translated image must be compiled with the /TIE qualifier and be linked with the /NONATIVE_ONLY qualifier. Checking for interoperability between native and translated images occurs at run time. If the /TIE and /NONATIVE_ONLY qualifiers are not used to compile and link the native routine, an error occurs at run time when the native routine and a translated image attempt to interoperate. If such an error occurs, recompile and relink the native routine appropriately.
- An access violation can occur at run time if a native routine that was not compiled with the /TIE qualifier makes an indirect call to a translated routine. The indirect call is made through a variable that contains the translated routine's address. When this happens, there is no autojacketing code in place to assist the native-to-translated call. The native code attempts to use the routine address as a native procedure descriptor. The code address of a native procedure is at offset PDSC$L_ENTRY, whose value is 8, from the base of the procedure descriptor. Because the translated routine address is treated as a procedure descriptor, the value at offset 8 from that address is used as the code to call. This usually results in an access violation.

  If you are encountering this problem, use a debugger to check the following:

  - Check that R27 points into a translated image.
  - Check that bits <31:2> of 8(R27) equal bits <31:2> of the access violation address. (All bits are not used because Alpha instructions are longword aligned.)
  - Check that R26 points into a native image.
  - Check that -4(R26) is a JSR R26,(26) instruction.

  If all these checks prove to be true, recompile the native routine with the /TIE qualifier to enable autojacketing at run time.

10.3.1.5 VAX C: Translated Program Restrictions

The following translated VAX C program restrictions are permanent:

- If a program uses the VAX C RTL routine brk( ) to release dynamic memory (that is, a break address lower than the current break address is requested), the next attempt by TIE to use a complex instruction routine may result in a fatal memory access violation. This may happen because the complex instruction routines are in a separate image, TIE$EMULAT_TV.EXE, which is dynamically activated by LIB$FIND_IMAGE_SYMBOL on the first use of one of the routines. Depending on when this occurs and the address passed to the brk( ) call that releases memory, the memory into which TIE$EMULAT_TV.EXE is loaded may also be released.

To avoid this problem, never use brk( ) to release memory, or be sure to execute a complex VAX instruction before getting the break address that is later used to release memory. Using brk( ) to allocate memory is fine.

- A translated VAX C program that uses vfork( ) and any executive function may hang at run time. If the child process of the VAX C program aborts erroneously, it may hang waiting for a mailbox I/O to be completed. One workaround is to prevent the child process from aborting.

10.4 Translated Image Support

At the beginning of the OpenVMS Alpha program, translation support was provided to remove impediments for users moving to Alpha due to:

- Lack of full language support initially
- Unavailability of source code for recompilation
- Difficulty of recompiling code that depended heavily on certain features of the VAX architecture

For languages whose VAX versions are undergoing active development, native Alpha versions are now available. The TIE is being maintained to support those language features that were available as of the release of OpenVMS VAX Version 5.5-2.

Similarly, translation is supported for images whose use of system services and run-time library entry points is restricted to those that existed on OpenVMS VAX Version 5.5-2.

In cases where more recent VAX layered products have been installed, it may be necessary to take minor additional steps if application needs require rebuilding an image suitable for translation. For instance, with DECwindows Motif Version 1.2 or Version 1.2–3 for OpenVMS VAX, images must be built with the OSF Motif Version 1.1.3 library or the DECwindows XUI library rather than with the OSF Motif Version 1.2.2 or Version 1.2.3 library in order to be suitable for translation.

Similarly, for those using recent versions of DEC Fortran for VAX, an additional qualifier is required to compile Fortran programs that are suitable for translation.

For further information, see the release notes for particular VAX products.

As a safety measure for situations where future rebuilding and retranslation of OpenVMS VAX images is likely, it may be preferable to save copies of the relevant OpenVMS VAX Version 5.5-2 shareable images in a separate VAX directory and link new versions of VAX applications against those images. Using that technique the resulting images will be compatible with newer OpenVMS VAX shareable images (picking up any OpenVMS enhancements of existing features), and will also be properly built for translation to OpenVMS Alpha (by not requiring newer versions of shareable images).

The following sections list the translated images, image information files, and other related files that are provided with OpenVMS Alpha.

OpenVMS Alpha contains no translated message images. All message images have been made native.

Translated Images in SYS$LIBRARY:

BASRTL2_D53_TV.EXE
BASRTL2_D56_TV.EXE
BASRTL_D56_TV.EXE
BASRTL_TV_SUPPORT.EXE
BLAS1RTL_D53_TV.EXE
BLAS1RTL_D56_TV.EXE
COBRTL_D56_TV.EXE
DBLRTL_D56_TV.EXE
EDTSHR_TV.EXE
FORRTL2_TV.EXE
FORRTL_D56_TV.EXE
LIBRTL2_D56_TV.EXE
LIBRTL_D56_TV.EXE
MTHRTL_D53_TV.EXE
MTHRTL_D56_TV.EXE
PASRTL_D56_TV.EXE
PLIRTL_D56_TV.EXE
RPGRTL_TV.EXE
SCNRTL_TV.EXE
TECOSHR_TV.EXE
TIE$EMULAT_TV.EXE
UVMTHRTL_D53_TV.EXE
UVMTHRTL_D56_TV.EXE
VAXCRTLG_D56_TV.EXE
VAXCRTL_D56_TV.EXE
VMSRTL_TV.EXE

Translated Images in SYS$SYSTEM:

DBLMSGMGR_TV.EXE
EDF_TV.EXE
EDT_TV.EXE
MONITOR_TV.EXE
TECO32_TV.EXE

Translated RTL Images in IMAGELIB:

BASRTL2_D53_TV.EXE
BASRTL_D56_TV.EXE
BLAS1RTL_D53_TV.EXE
COBRTL_D56_TV.EXE
DBLRTL_D56_TV.EXE
FORRTL2_TV.EXE
FORRTL_D56_TV.EXE
LIBRTL_D56_TV.EXE
PLIRTL_D56_TV.EXE
RPGRTL_TV.EXE
SCNRTL_TV.EXE

Note that most of the translated RTLs are provided in D56 format rather than D53 format; some are provided in both formats. Where both formats are provided, the default format is D53. See Section 10.5 for more information about the translated run-time libraries.

Image Information Files in SYS$LIBRARY:

ACLEDTSHR.IIF
BASRTL2.IIF
BASRTL.IIF
BLAS1RTL.IIF
COBRTL.IIF
CONVSHR.IIF
CRFSHR.IIF
DBLRTL.IIF
DCXSHR.IIF
DISMNTSHR.IIF
DTKSHR.IIF
EDTSHR.IIF
ENCRYPSHR.IIF
EPC$SHR.IIF
FDLSHR.IIF
FORRTL.IIF
FORRTL2.IIF
INIT$SHR.IIF
LBRSHR.IIF
LIBRTL.IIF
LIBRTL2.IIF
MAILSHR.IIF
MOUNTSHR.IIF
MTHRTL.IIF
NCSSHR.IIF
P1_SPACE.IIF
PASRTL.IIF
PLIRTL.IIF
PPLRTL.IIF
PTD$SERVICES_SHR.IIF
RPGRTL.IIF
S0_SPACE.IIF
SCNRTL.IIF
SCRSHR.IIF
SECURESHR.IIF
SMBSRVSHR.IIF
SMGSHR.IIF
SORTSHR.IIF
SPISHR.IIF
TECOSHR.IIF
TPUSHR.IIF
UVMTHRTL.IIF
VAXCRTL.IIF
VAXCRTLG.IIF
VMSRTL.IIF

System Logical Names Definitions

The following system logical names are defined to facilitate the translated environment:

ACLEDTSHR_TV = ACLEDTSHR
CDDSHR_TV = CDDSHR
CONVSHR_TV = CONVSHR
CRFSHR_TV = CRFSHR

```
DCXSHR_TV = DCXSHR
DISMNTSHR_TV = DISMNTSHR
DTKSHR_TV = DTKSHR
ENCRYPSHR_TV = ENCRYPSHR
EPC$SHR_TV = EPC_SHR
FDLSHR_TV = FDLSHR
INIT$SHR_TV = INIT$SHR
LBRSHR_TV = LBRSHR
MAILSHR_TV = MAILSHR
MOUNTSHR_TV = MOUNTSHR
NCSSHR_TV = NCSSHR
PPLRTL_TV = PPLRTL
PTD$SERVICES_SHR_TV = PTD$SERVICES_SHR
SCRSHR_TV = SCRSHR
SECURESHR_TV = SECURESHR_JACKET
SMBSRVSHR_TV = SMBSRVSHR
SMGSHR_TV = SMGSHR
SORTSHR_TV = SORTSHR
SPISHR_TV = SPISHR
TPUSHR_TV = TPUSHR

BASRTL_TV = BASRTL_D56_TV
BASRTL2_TV = BASRTL2_D53_TV
BLAS1RTL_TV = BLAS1RTL_D53_TV
COBRTL_TV = COBRTL_D56_TV
DBLRTL_TV = DBLRTL_D56_TV
FORRTL_TV = FORRTL_D56_TV
LIBRTL_TV = LIBRTL_D56_TV
LIBRTL2_TV = LIBRTL2_D56_TV
MTHRTL_TV = MTHRTL_D53_TV
PASRTL_TV = PASRTL_D56_TV
PLIRTL_TV = PLIRTL_D56_TV
VAXCRTL_TV = VAXCRTL_D56_TV
VAXCRTLG_TV = VAXCRTLG_D56_TV

DBLMSGMGR = DBLMSGMGR_TV
EDTSHR_TV = EDTSHR
TECO32 = TECO32_TV
TECOSHR = TECOSHR_TV
VMSRTL = VMSRTL_TV

DBLRTLMSG = DBL$MSG
PASMSG = PAS$MSG
PLIMSG = PLI$MSG
RPGMSG = RPG$MSG
SCNMSG = SCN$MSG
VAXCMSG = DECC$MSG
```

10.5 Translated Run-Time Libraries

As part of the OpenVMS Alpha kit, Digital provides a set of translated run-time libraries.

Some of the routines in the VAX run-time libraries use the VAX D_floating data type for double-precision arithmetic.

In the translated versions of these libraries, the Alpha D56 D_floating data type is used by default (where the VAX run-time library used D_floating). This provides the full precision of the 56-bit mantissa in VAX D_floating, yielding consistency of results at a cost in execution-time performance.

For a handful of performance-critical math-related libraries, Digital also supplies versions of the translated run-time libraries that use the Alpha D53 D_floating data type for double-precision operations. For these libraries, the D53 forms are the default. The D53 forms provide better performance by sacrificing the low-order three bits of precision in the mantissa.

The following translated libraries are provided in D56 form only:

- BASRTL
- COBRTL
- DBLRTL
- FORRTL
- LIBRTL
- LIBRTL2
- PASRTL
- PLIRTL
- VAXCRTL
- VAXCRTLG

The following translated libraries are provided in both D56 and D53 (the default) form:

- BASRTL2
- BLAS1RTL
- MTHRTL
- UVMTHRTL

Accessing the D56 Form of the Run-Time Libraries

When you use the run-time libraries, the following happens by default:

- For BASRTL2, translated BASIC images that use MAT functions on double-precision data invoke BASIC run-time library routines that use the D53 data type.
- For BLAS1RTL, translated images that invoke BLAS$ functions with double-precision floating-point arguments get routines that use the D53 data type.
- For MTHRTL, translated images that invoke MTH$ double-precision floating-point functions get routines that use the D53 data type.
- For all others, the Alpha D56 floating-point data type is used by default.

Some users might need the full precision of D56 floating point. However, using the D56 routines imposes a very significant performance penalty. To access the D56 routines, redefine the run-time library's logical name to the D56 form, as shown in Table 10–3. The logical name can be defined on a per-process or systemwide basis, as appropriate for your site.

Table 10–3 Run-Time Library Logical Names

Library	Logical Name	D56 Name
BASRTL2	BASRTL2_TV	BASRTL2_D56_TV
BLAS1RTL	BLAS1RTL_TV	BLAS1RTL_D56_TV
MTHRTL	MTHRTL_TV	MTHRTL_D56_TV

10.5.1 CRF$FREE_VM and CRF$GET_VM: Translated Callers

Translated callers to CRF$FREE_VM and CRF$GET_VM will not work properly. The translated callers are expecting VAX JSB semantics, but instead, Alpha JSB semantics are present in the native code (naturally).

To work around this problem, the translated callers need to use CALL instead of JSB.

10.6 Translated VAX C Run-Time Library

The following sections contain release notes pertaining to the translated VAX C run-time library.

10.6.1 Problems and Restrictions

This section describes known problems and restrictions with the OpenVMS Alpha translated VAX C Run-Time Library (VAX C RTL).

10.6.1.1 Functional Restrictions

The translated VAX C RTL is a translated version of the OpenVMS VAX Version 5.4 VAX C RTL. All problems and restrictions present in that release of the VAX C RTL exist unchanged in the translated VAX C RTL. The following items are known restrictions in the functionality of the translated VAX C RTL:

- The fmod( ) function does not produce correct results for D_FLOAT.
- D_FLOAT programs that use the SIGFPE signal may not catch all floating-point exceptions. Translating the program using /FLOAT=D56_FLOAT fixes most SIGFPE problems.
- The sbrk( ) function returns an address that does not match the value returned from SYS$EXPREG.
- D_FLOAT programs that use the HUGE_VAL constant or call the math functions (which may return HUGE_VAL) may fail unless they are translated with /FLOAT=D56_FLOAT.
- Under certain circumstances, some math functions (either D_FLOAT or G_FLOAT) may generate a high-performance arithmetic trap exception instead of setting errno to ERANGE or EDOM.

10.6.1.2 Interoperability Restrictions

The following restrictions apply when the translated VAX C RTL interoperates with the native DEC C RTL:

- The longjmp function cannot be used to transfer control from:
  - A native routine to a translated routine
  - A translated routine to a native routine
- Memory allocated by malloc, calloc, and so forth must be freed in the same context. That is, if a translated routine allocates memory, the free call must occur in a translated routine. Allocating memory in a translated routine and freeing it in a native routine results in corruption of the heap. Likewise, allocating memory in a native routine and freeing that memory in a translated routine also corrupts the heap.
- Signal handlers established by the signal (and related) functions in translated routines are not invoked when the signal is raised. Only native signal handlers can be used to catch UNIX style signals.
- The signals SIGEMT, SIGTRAP, SIGIOT, and SIGFPE cannot be caught if those signals are raised by a translated image.
- The exec function can be used only to invoke similar images. That is, an exec function invoked in a native image cannot execute a translated image. Likewise, an exec function invoked in a translated image cannot execute a native image.
- An access violation occurs if vfork is executed in a native image to establish the context for a later system call and the system call is then invoked in a translated image.
- File pointers and file descriptors cannot be shared between native and translated images. An access violation or file corruption is likely to occur if a file is opened in a translated image and a native image attempts to read or write using that file pointer. The same results occur if a file is opened in a native image and a translated image attempts to read or write using that file pointer.

Programs that perform any of these restricted actions may receive access violations or other exceptions. No testing is performed to detect and prevent restricted operations from being performed.

10.7 Translated VAX COBOL Programs

The OpenVMS Alpha operating system supports the execution of translated VAX COBOL programs compiled with the VAX COBOL Version 5.0 compiler (or earlier compilers).

10.7.1 Problems and Restrictions

Programs compiled with the VAX COBOL Version 5.1 compiler are not supported by the OpenVMS Alpha operating system.

11
Ensuring Interoperability Between Native and Translated Images

This chapter describes how to create native Alpha images that can make calls to and be called by translated VAX images.

11.1 Overview

DECmigrate for OpenVMS AXP Systems Translating Images describes how to use the VAX Environment Software Translator (VEST) to convert a VAX executable or shareable image into a functionally equivalent Alpha image. (DECmigrate for OpenVMS Alpha is an optional layered product that supports the migration of a VAX application to an Alpha system. VEST is a component of the DECmigrate utility.)

Using VEST, you can translate all the components of an application, such as the main executable image and all the shareable images that it calls. However, you can also create an application that is a mix of translated and native components. For example, you may want to create a native version of a shareable image that is called by your application to take advantage of native performance. You may also choose to use a mixture of native and translated components to allow you to create a native version of your application in stages.

You can use translated VAX images as you would a native Alpha image. However, to create native images that can interoperate with translated images requires some additional considerations, described in the following sections.

11.1.1 Compiling Native Images That Can Interoperate with Translated Images

To create a native image that can call or be called by a translated image, you must specify the /TIE qualifier when compiling the source files of the native Alpha image. Any source module that contains a procedure that is made available to external callers must be compiled with the /TIE qualifier. When you specify the /TIE qualifier, the compilers create procedure signature blocks (PSBs) that are needed by the Translated Image Environment (TIE) at execution time in order to properly jacket calls between translated and native images. The TIE is part of the operating system.

You must also specify the /TIE qualifier when compiling a source module that contains a procedure that performs a callback (or calls out to a specified procedure) that may be in a translated image. In this case, the /TIE qualifier causes the compilers to generate a call to a special run-time library routine, OTS$CALL_PROC, that ensures that the outbound call to a translated procedure is handled properly.

In addition to the /TIE qualifier, you may need to specify other compiler qualifiers to ensure correct interoperation between a translated image and a native shareable image. For example, if the translated callers of a native shareable image use the VAX D_floating format for double-precision floating-point operations (the default for VAX languages), you must specify the /FLOAT=D_FLOAT qualifier because the default format for double-precision data on Alpha systems is *not* VAX D_floating. Consult compiler documentation to determine the exact qualifier syntax to specify VAX D_floating format. Note that, because the VAX D_floating data type is not supported by the Alpha architecture, its use adversely affects performance.

Depending on application-specific semantics, you may also need to specify other compiler qualifiers to force byte granularity, data alignment, and AST atomicity.

11.1.2 Linking Native Images That Can Interoperate with Translated Images

To create a native Alpha image that can call a translated VAX image, you must explicitly link the native object modules with the /NONATIVE_ONLY qualifier. (Note that /NATIVE_ONLY is the default used by the linker for this qualifier.) This qualifier causes the linker to include in the image the PSB information created by the compilers.

Because the /NONATIVE_ONLY qualifier affects only outgoing calls from native images to translated images, you do not need to specify it when creating a native Alpha image that will be called by a translated VAX image. The linker's default behavior (/NATIVE_ONLY qualifier) can prevent native images from calling translated images but not from being called by translated images.

Note that the layout of the symbol vector in the native version of the shareable image must match the layout of the symbol vector in the translated shareable image it replaces. For more information about replacing translated shareable images with native shareable images, see Section 11.3.

11.2 Creating a Native Image That Can Call a Translated Image

To create a native Alpha image that can make calls to a translated VAX shareable image, perform the following steps:

1. **Translate the VAX shareable image**. See *DECmigrate for OpenVMS AXP Systems Translating Images* for information about using VEST to translate VAX images.
2. **Create a native Alpha version of the main program**. Compile the source modules using a compiler that produces native Alpha code, specifying the /TIE qualifier on the command line.
3. **Link the native object module with the translated VAX shareable image**. Specify the translated image in a linker options file as you would any other shareable image.

To illustrate interoperability, consider the programs in Example 11–1 and Example 11–2. Example 11–1 calls the routine mysub that is defined in Example 11–2.

Example 11–1 Source Code for Main Program (MYMAIN.C)

```
#include <stdio.h>

int mysub();

main()
{
   int num1, num2, result;

   num1 = 5;
   num2 = 6;

   result = mysub( num1, num2 );
   printf("Result is: %d\n", result);
}
```

Example 11–2 Source Code for Shareable Image (MYMATH.C)

```
int myadd(value_1, value_2)
 int value_1;
 int value_2;
{
  int result;

  result = value_1 + value_2;
  return( result );
}

int mysub(value_1,value_2)
 int value_1;
 int value_2;
{
 int result;

 result = value_1 - value_2;
 return( result );
}

int mydiv( value_1, value_2 )
  int value_1;
  int value_2;
{
  int result;

  result = value_1 / value_2;
  return( result );
}

int mymul( value_1, value_2 )
  int value_1;
  int value_2;
{
  int result;

  result = value_1 * value_2;
  return( result );
}
```

To create VAX images from these examples, compile the source modules using a C compiler on a VAX system. To implement Example 11–2 as a shareable image, link the module, specifying the /SHAREABLE qualifier on the LINK command line and declaring the universal symbols in the shareable image by using the UNIVERSAL= option or by creating a transfer vector file. (See the Bookreader version of the *OpenVMS Linker Utility Manual* for information about how to create a shareable image.) The following example shows a LINK command that creates the shareable image MYMATH.EXE:

```
$ LINK/SHAREABLE  MYMATH.OBJ,SYS$INPUT/OPT
GSMATCH=LEQUAL,1,1000
UNIVERSAL=myadd
UNIVERSAL=mysub
UNIVERSAL=mydiv
UNIVERSAL=mymul
Ctrl/Z
```

You can then link the main program with the shareable image to create the executable image MYMAIN.EXE, as in the following example:

```
$ LINK  MYMAIN.OBJ,SYS$INPUT/OPT
MYMATH.EXE/SHAREABLE
Ctrl/Z
```

Note that you may need to specify the /BPAGE qualifier on the LINK command line to force the linker to create image sections using a larger page size than the default page size on VAX systems (512 bytes). Otherwise, when VEST translates your VAX image, VEST may collect a number of these 512-byte image sections on a single Alpha page. When VEST puts neighboring image sections with conflicting protection attributes on the same Alpha page, it assigns the most permissive protection to all the image sections and issues a warning. (See the Bookreader version of the *OpenVMS Linker Utility Manual* for more information about using the /BPAGE qualifier.)

After creating the VAX images, translate them using VEST. Note that you must translate the shareable image first. (For more information about using the VEST command, see *DECmigrate for OpenVMS AXP Systems Translating Images*.) The following example creates the translated files named MYMATH_TV.EXE and MYMAIN_TV.EXE (VEST appends the characters "_TV" to the end of the image's file name):

```
$ VEST  MYMATH.EXE
$ VEST  MYMAIN.EXE
```

To replace the translated main executable image MYMAIN_TV.EXE with a native version, compile the source module in Example 11–1 using a compiler that generates Alpha code, specifying the /TIE qualifier on the compile command line. Then link the native object module MYMAIN.OBJ to create a native Alpha image, specifying the translated shareable image in the linker options file as you would any other shareable image, as in the following example:

```
$ LINK/NONATIVE_ONLY  MYMAIN.OBJ,SYS$INPUT/OPT
MYMATH_TV.EXE/SHAREABLE
Ctrl/Z
```

You can run the native main image as you would any other Alpha image. Define the name of the translated shareable image, MYMATH_TV, as a logical name that points to the location of the translated shareable image (unless it is located in the directory pointed to by the SYS$SHARE logical name) and execute the RUN command, as in the following example:

```
$ DEFINE MYMATH_TV YOUR$DISK:[YOUR_DIR]MYMATH_TV.EXE
$ RUN MYMAIN
```

11.3 Creating a Native Image That Can Be Called by a Translated Image

To create a native Alpha shareable image that can be called by a translated VAX image, perform the following steps:

1. **Translate the VAX shareable image**. Even though you are replacing the VAX version of the shareable image with a native version, you must still translate the shareable image to create a VEST interface information file (IIF). VEST needs the IIF associated with the shareable image when it translates an image that calls the shareable image. See *DECmigrate for OpenVMS AXP Systems Translating Images* for information about IIF files and about using VEST to translate VAX images. (Note that you may have to repeat this step to control the layout of the symbol vector in the translated shareable image. See Section 11.3.1 for more information.)
2. **Translate the VAX executable image that calls the shareable image**.
3. **Create a native Alpha version of the shareable image**. Compile the source modules using a compiler that generates Alpha code, specifying the /TIE qualifier on the command line.
4. **Link the object module to create a native Alpha shareable image**. Use the SYMBOL_VECTOR= option to declare the universal symbols in the shareable image. For compatibility, declare the universal symbols in the SYMBOL_VECTOR= option in the same order as they were declared in the VAX shareable image.

Note

When creating a native Alpha shareable image to replace a translated VAX shareable image, always leave the first entry of a symbol vector empty by specifying the SPARE keyword as the first entry in the SYMBOL_VECTOR= option. VEST reserves the first symbol vector entry in the translated VAX image for its own use.

The following example creates a native shareable image from the source module in Example 11–2:

```
$ LINK/SHAREABLE  MYMATH.OBJ,SYS$INPUT/OPT
GSMATCH=LEQUAL,1,1000  ❶
SYMBOL_VECTOR=(SPARE,-
               myadd=procedure,-  ❷
               mysub=procedure,-
               mydiv=procedure,-
               mymul=procedure)
Ctrl/Z
```

❶ Specifies the major and minor identification numbers of the shareable image. The values of these identification numbers must match the values specified in the VAX shareable image. (For more information about using the GSMATCH= option, see the Bookreader version of the *OpenVMS Linker Utility Manual*.)

❷ Specifies the universal symbols in the shareable image.

5. **Make sure the layout of the symbol vector in the native Alpha image matches the symbol vector in the translated VAX image**. Section 11.3.1 describes how to determine the offsets of symbols in these symbol vectors and how to control the layout of these symbol vectors to ensure they match.

You can run the translated main image, MYMAIN_TV.EXE, with either the translated VAX shareable image, MYMATH_TV.EXE, or with the native Alpha shareable image, MYMATH.EXE. By default, the translated executable image calls the translated shareable image. (The translated executable image contains a global image section descriptor [GISD] that points to the translated shareable image. The image activator activates the shareable images to which the image has been linked.)

To run the translated main image with the native shareable image, define the name of the shareable image MYMATH_TV as a logical name that points to the location of the native Alpha shareable image, MYMATH.EXE, as in the following example:

```
$ DEFINE MYMATH_TV YOUR_DISK:[YOUR_DIR]MYMATH.EXE
$ RUN MYMAIN_TV
```

11.3.1 Controlling Symbol Vector Layout

To create a native Alpha shareable image that can replace a translated VAX shareable image in an application, you must ensure that the universal symbols in the shareable images appear at the same offsets in the symbol vectors in both images. When a VAX shareable image is translated, VEST creates a symbol vector for the image that includes the universal symbols declared in the original VAX shareable image. (A translated image is actually an Alpha image, created by VEST, and, like any other Alpha shareable image, it lists universal symbols in a symbol vector.) To create a native shareable image that is compatible with a translated shareable image, you must make sure that the same symbols appear at the same offsets in the symbol vector in the native Alpha shareable image and in the translated VAX shareable image it replaces.

To control how VEST lays out the symbol vector it creates in the translated VAX image, create a symbol information file (SIF) and include it in the translation operation. An SIF is a text file with which you can specify the layout of entries in the symbol vector VEST creates for the translated image and to determine which symbols should appear in the global symbol table (GST) of the translated shareable image. If you do not specify the layout of the symbol vector, VEST may change the layout in subsequent retranslations of the shareable image. Note that VEST reserves the first symbol vector entry for its own use. For more information about SIFs, see *DECmigrate for OpenVMS AXP Systems Translating Images*.

You control the layout of the symbol vector in a native shareable image by specifying the SYMBOL_VECTOR= option. The linker lays out the entries in a symbol vector in the order in which you specify the symbols in the SYMBOL_VECTOR= option statement. Make sure you list the symbols in the SYMBOL_VECTOR= option in the same order as they appear in the transfer vector used to create the VAX shareable image. For more information about using the SYMBOL_VECTOR= option, see the Bookreader version of the *OpenVMS Linker Utility Manual*.

To make sure the symbol vector in a translated shareable image matches the symbol vector in a native shareable image, perform the following steps:

1. **Translate the VAX shareable image, specifying the /SIF qualifier**. When you specify the /SIF qualifier, VEST generates an SIF that lists the contents of the symbol vector. (For more information about creating and interpreting an SIF, see *DECmigrate for OpenVMS AXP Systems Translating Images*.) The following example is the SIF created by VEST for the shareable image MYMATH.EXE. Note that the entries start at the *second* position in the symbol vector (offset 10 hexadecimal):

```
! .SIF Generated by VEST (V1.0) on
! Image "MYMATH", "V1.0"
MYDIV                           00000018 +S +G 00000030    00 4e
MYSUB   ➊                       0000000c +S +G 00000020 ➋ 00 4e
MYADD                           00000008 +S +G 00000010    00 4e
MYMUL                           00000010 +S +G 00000040    00 4e
```

   ➊ The entry for the universal symbol MYSUB.

   ➋ The offset of the entry for MYSUB in the translated image's symbol vector.

2. **Determine the symbol vector offsets in the native shareable image.** Use the ANALYZE/IMAGE utility to determine the offsets of the symbols in the symbol vector in the native shareable image. The following excerpt from an analysis of the shareable image MYMATH.EXE shows the offset of the symbol MYSUB:

```
   .
   .
   .
 4) Universal Symbol Specification (EGSD$C_SYMG)
  data type: DSC$K_DTYPE_Z (0)
  symbol flags:
   (0)  EGSY$V_WEAK      0
   (1)  EGSY$V_DEF       1
   (2)  EGSY$V_UNI       1
   (3)  EGSY$V_REL       1
   (4)  EGSY$V_COMM      0
   (5)  EGSY$V_VECEP     0
   (6)  EGSY$V_NORM      1
  psect: 0
  value: 16 (%X'00000020')
  symbol vector entry (procedure)
   %X'00000000 00010000'
   %X'00000000 00000050'
  symbol: "MYSUB"
   .
   .
   .
```

3. **Edit the offsets in the SIF, if necessary**. Use a text editor to change the offsets listed in the SIF if they do not match the offsets in the native shareable image. Remember that the first entry in the symbol vector is reserved for use by the VEST utility.

4. **Retranslate the VAX shareable image, including the SIF in the translation operation.** In this translation operation, VEST creates the symbol vector in the translated image using the offsets specified in the SIF. VEST looks for the SIF in the current device and directory. (See *DECmigrate for OpenVMS AXP Systems Translating Images* for more information about using VEST.)

11.3.2 Creating Stub Images

In some cases, it is not possible to completely replace a VAX shareable image with an Alpha shareable image. For example, there may be functions in the VAX shareable image that are specific to the VAX architecture. In this situation, it may be necessary to build both a translated image and a native image that together provide the functionality of the original VAX shareable image. In other cases, there may be a need to define a relationship between a translated VAX shareable image and a new Alpha shareable image. In both situations, the translated VAX image must be a **jacket image**.

When building a jacket image, create a stub version of the new Alpha image on a VAX system. Then create a modified VAX shareable image that depends on it and translate it, specifying the /JACKET=*shrimg* qualifier, where *shrimg* is the name of the new Alpha shareable image. Note that a throwaway translation of the stub image must be performed in advance so that there is an IIF that describes it. For complete information about creating stub images, see *DECmigrate for OpenVMS AXP Systems Translating Images*.

Part III

Layered Products

12
OpenVMS Alpha Compilers

This appendix provides information about the features that are specific to the native OpenVMS Alpha compilers. In addition, it lists the features of the OpenVMS VAX compilers that are not supported by or that have changed behavior in their OpenVMS Alpha counterparts.

The following compilers are covered in this appendix:

- DEC Ada (Section 12.1)
- DEC C (Section 12.2)
- DEC COBOL (Section 12.3)
- DEC Fortran and Digital Fortran (Alpha systems only) (Section 12.4)
- DEC Pascal (Section 12.5)

Compiler differences can exist for two reasons: differences between earlier and current versions of compilers running on OpenVMS VAX and differences between the DEC versions running on the VAX and Alpha computers. The OpenVMS Alpha compilers are intended to be compatible with their OpenVMS VAX counterparts. They include several qualifiers that contribute to compatibility, as described in the following sections.

The languages conform to language standards and include support for most OpenVMS VAX language extensions. The compilers produce output files with the same default file types as they do on OpenVMS VAX systems, such as .OBJ for an object module.

Note, however, that some features supported by the compilers on OpenVMS VAX systems may not be available on OpenVMS Alpha systems.

For more information about the compiler differences for each language, refer to its documentation, especially the user's guides and the release notes.

12.1 Compatibility of DEC Ada Between Alpha Systems and VAX Systems

DEC Ada includes nearly all the standard and extended Ada language features included in VAX Ada. These features are documented in the following manuals:

- *DEC Ada Language Reference Manual*
- *Developing Ada Programs on OpenVMS Systems*
- *DEC Ada Run-Time Reference Manual for OpenVMS Systems*

However, owing to differences in the platform hardware, some features are not supported or are implemented differently on VAX systems than on Alpha systems. To aid in porting programs from one system to another, the following sections highlight these differences.

Note

Not all of these features may be implemented on all systems for each release. See the DEC Ada release notes for more information.

12.1.1 Differences in Data Representation and Alignment

In general, DEC Ada supports the same data types on all platforms. However, keep in mind the following differences:

- H_floating data

  Supported on VAX systems but not supported on Alpha systems.

- IEEE floating-point formats

  Supported on Alpha systems but not supported on VAX systems.

- Natural alignment

  On Alpha systems, DEC Ada aligns record and array components on natural boundaries by default. On VAX systems, DEC Ada aligns record and array components on byte boundaries. Note that you can specify the alignment with the pragma COMPONENT_ALIGNMENT. The record representation clause maximum alignment is 2^9 on both VAX and Alpha systems.

12.1.2 Tasking Differences

Task priorities and scheduling and task control block size are architecture specific. See the release notes for specifics.

12.1.3 Differences in Language Pragmas

Note the following differences in language pragmas:

- pragma COMPONENT_ALIGNMENT

  On Alpha systems, COMPONENT_SIZE is the default choice. On VAX systems, STORAGE_UNIT is the default.

- pragma FLOAT_REPRESENTATION

  On Alpha systems, this pragma supports two choices, IEEE_FLOAT and VAX_FLOAT. On VAX systems, this pragma supports the VAX_FLOAT choice.

- pragma LONG_FLOAT

  On Alpha systems, the LONG_FLOAT pragma is supported when the value of the FLOAT_REPRESENTATION pragma is VAX_FLOAT.

- pragma SHARED

  There are type restrictions that are different between the systems. See the *DEC Ada Run-Time Reference Manual for OpenVMS Systems* for more information.

- pragma MAIN_STORAGE

  Not supported on Alpha systems.

- pragma SHARE_GENERIC

  Not supported on Alpha systems.

- pragma TIME_SLICE

  There are some implementation differences between the support of this pragma on VAX systems and on Alpha systems. See the *DEC Ada Run-Time Reference Manual for OpenVMS Systems* for more information.

12.1.4 Differences in the SYSTEM Package

Note the following changes to the system package:

- SYSTEM.IEEE_SINGLE_FLOAT and SYSTEM.IEEE_DOUBLE_FLOAT

  Supported on Alpha systems but not on VAX systems.

- SYSTEM.H_FLOAT

  Supported on VAX systems but not on Alpha systems.

- SYSTEM.MAX_DIGITS

  The value is 33 on VAX systems and 15 on Alpha systems.

- SYSTEM.NAME

  Specific numerals are supported for each system on which DEC Ada is available.

- SYSTEM.SYSTEM_NAME

  The name OpenVMS_Alpha is supported on Alpha systems.

- SYSTEM.TICK

  The value is 10.0^{-3} (1 ms) on Alpha systems. The value on VAX systems is 10.0^{-2} (10 ms).

In addition, the following types and subprograms supported on VAX systems are not supported on Alpha systems:

SYSTEM.READ_REGISTER
SYSTEM.WRITE_REGISTER
SYSTEM.MFPR
SYSTEM.MTPR
SYSTEM.CLEAR_INTERLOCKED
SYSTEM.SET_INTERLOCKED
SYSTEM.ALIGNED_WORD
SYSTEM.ADD_INTERLOCKED
SYSTEM.INSQ_STATUS
SYSTEM.REMQ_STATUS
SYSTEM.INSQHI
SYSTEM.REMQHI
SYSTEM.INSQTI
SYSTEM.REMQTI

12.1.5 Differences Between Other Language Packages

Note the following differences in these other packages:

- package CALENDAR

  Implementation differences between systems; see the *DEC Ada Language Reference Manual* for more information.

- package MATH_LIB

  Implementation differences between systems; see individual package specifications.

- package SYSTEM_RUNTIME_TUNING

  This package is supported on VAX systems and, with some restrictions, on Alpha systems. See the *DEC Ada Run-Time Reference Manual for OpenVMS Systems* or the release notes for more information.

12.1.6 Changes to Predefined Instantiations

The following two predefined instantiations supported on VAX systems are not supported on Alpha systems:

- LONG_LONG_FLOAT_TEXT_IO
- LONG_LONG_FLOAT_MATH_LIB

12.2 Compatibility of DEC C for OpenVMS Alpha Systems with VAX C

The DEC C compiler defines a core, ANSI-conforming C language that can be used on all strategic Digital platforms, including the Alpha architecture. For comprehensive information, see the DEC C documentation.

12.2.1 Language Modes

DEC C for OpenVMS Alpha systems conforms to the ANSI C standard, with optional support for VAX C and Common C (pcc) extensions. You invoke these optional extensions, called **modes**, using the /STANDARD qualifier. Table 12–1 describes these modes and the command-qualifier syntax needed to invoke them.

Table 12–1 Modes of Operation of the DEC C for OpenVMS Alpha Systems

Mode	Command Qualifier	Description
Default	/STANDARD=RELAXED_ANSI89	Follows ANSI C standard but also allows additional Digital keywords and predefined macros that do not begin with an underscore.
ANSI C	/STANDARD=ANSI89	Accepts only strictly conforming ANSI C language.
VAX C	/STANDARD=VAXC	Allows use of VAX C extensions to the ANSI C standard, even where the extensions may be incompatible with the ANSI C standard.
Common C (pcc)	/STANDARD=COMMON	Allows use of Common C extensions to the ANSI C standard, even where the extensions may be incompatible with the ANSI C standard.

12.2.2 DEC C for OpenVMS Alpha Systems Data-Type Mappings

The DEC C for OpenVMS Alpha systems compiler supports most of the same data-type mappings as its VAX counterpart. Table 12–2 lists the sizes of the C arithmetic data types on the Alpha architecture.

Table 12–2 Arithmetic Data-Type Sizes in DEC C for OpenVMS Alpha Compiler

C Data Type	VAX C Mapping	DEC C Mapping
pointer	32	32 or 64[1]
long	32	32
int	32	32
short	16	16
char	8	8
float	32	32[2]
double	64[2]	128[2]
long double	64[2]	64[2]
__int16	NA	16
__int32	NA	32
__int64	NA	64

[1]You select the size by using a pragma in your source file or by using a command line qualifier. For more information, see the *DEC C User's Guide for OpenVMS Systems*.

[2]You select how this maps to an Alpha D, F, G, S, T,or X floating point by using a command line qualifier. See Section 12.2.2.1.

To aid portability, the DEC C for OpenVMS Alpha compiler provides a header file that defines typedefs for the signed and unsigned variants of these data types. For example, if you have a data type that is a 64-bit integer, use the int64 typedef.

12.2.2.1 Specifying Floating-Point Mapping

The mapping between the C floating-point data types and the Alpha floating-point data types is controlled by command line qualifiers. The Alpha architecture supports the following floating-point types:

- F_floating (same as on OpenVMS VAX systems)
- D_floating (53-bit precision)
- G_floating (same as on OpenVMS VAX systems)
- S_floating (IEEE single precision–32 bits)
- T_floating (IEEE double precision–64 bits)
- X_floating (IEEE extended double precision–128 bits)

By using a command line qualifier, you control which of the Alpha floating-point data types the standard C data types float and double map to. For example, if you specify the /FLOAT=G_FLOAT qualifier, DEC C maps the float data type to the Alpha F_floating data type and maps the double data type to the Alpha G_floating data type. Table 12–3 describes the complete list of floating-point options. Note that you can specify only one floating-point qualifier in a command line.

Table 12–3 DEC C Floating-Point Mappings

Compiler Option	Float	Double	Long Double
/FLOAT=F_GLOAT	F_floating format	G_floating format	
/FLOAT=D_FLOAT	F_floating format	D-53 floating point	
/FLOAT=IEEE_FLOAT	S_floating format	T_floating format	
/L_DOUBLE_SIZE=128 (default)	—	—	X_floating format

12.2.3 Built-in Functions That Access Alpha Instructions

DEC C includes features, listed in Table 12–4, that are specific to Alpha systems. The following sections describe these features.

Table 12–4 DEC C Compiler Features Specific to Alpha Systems

Feature	Description
Access to some Alpha instructions	Available as built-ins
Access to some VAX instruction equivalents	Available through Alpha PALcode
Atomicity built-ins	Ensures the atomicity of AND, OR, and ADD operations

12.2.3.1 Accessing Alpha Instructions

DEC C supports certain Alpha instructions to provide functions that cannot be expressed in the C language, especially for system-level programming; for example:

- TRAPB—Drain the instruction pipeline
- MB—Memory barrier

12.2.3.2 Accessing Alpha Privileged Architecture Library (PALcode) Instructions

The Alpha architecture implements certain VAX instructions as privileged architecture library (PALcode) instructions. DEC C allows access to the following PALcode instructions:

- INSQUEx—Insert entry into longword or quadword queue
- INSQxI—Insert entry in queue, interlocked
- REMQUEx—Remove entry from longword or quadword queue
- REMQxI—Remove from queue, interlocked

12.2.3.3 Ensuring the Atomicity of Combined Operations

In the VAX architecture, certain combined operations, such as incrementing a variable, are guaranteed to be atomic (that is, they complete without interruption). To provide an equivalent function on Alpha systems, DEC C provides built-ins that perform the operations with the guarantee of atomicity. For example, several of these atomic built-ins are listed in Table 12–5. For a complete description of these built ins, see the DEC C language documentation.

Table 12–5 Atomicity Built-Ins

Atomicity Built-In	Description
__ADD_ATOMIC_LONG(ptr, expr, retry_count) __ADD_ATOMIC_QUAD(ptr, expr, retry_count)	Add the expression **expr** to the data segment pointed to by **ptr**. The optional **retry_count** parameter specifies the number of times the operation should be attempted (the default is forever).
__AND_ATOMIC_LONG(ptr, expr, retry_count) __AND_ATOMIC_QUAD(ptr, expr, retry_count)	Fetch the data segment pointed to by **ptr**, perform a logical AND operation with the expression **expr**, and store the resulting value. The **retry_count** parameter specifies the number of times the operation should be attempted (the default is forever).
__OR_ATOMIC_LONG(ptr, expr, retry_count) __OR_ATOMIC_QUAD(ptr, expr, retry_count)	Fetch the data segment pointed to by **ptr**, perform a logical OR operation with the expression **expr**, and store the resulting value. The **retry_count** parameter specifies the number of times the operation should be attempted (the default is forever).

These built-ins guarantee only that the operation completes without interruption. If you perform an atomic operation on a variable that might be subject to concurrent write access (for example, from an AST and mainline code or from two concurrent processes), you must still protect it with the **volatile** attribute.

In addition, DEC C for OpenVMS Alpha systems supports the following equivalents to the VAX interlocked instructions:

- TESTBITSSI
- TESTBITCCI

These built-ins use the **retry_count** parameter, as do the atomicity built-ins, to avoid getting stuck in an endless loop.

12.2.4 Differences Between the VAX C and DEC C for OpenVMS Alpha Systems Compilers

The following features, present in VAX C, have different default behavior in DEC C for OpenVMS Alpha systems. Note, however, that for some of these features, you can retain the VAX C behavior by using command line qualifiers and pragma instructions.

12.2.4.1 Controlling Data Alignment

Because accesses to data that is not aligned on natural boundaries cause severe performance degradation on Alpha systems, DEC C for OpenVMS Alpha systems aligns data on natural boundaries by default. To override this feature and retain VAX (packed) alignment, specify the nomember_alignment pragma in your source file or use the /NOMEMBER_ALIGNMENT command line qualifier.

12.2.4.2 Accessing Argument Lists

Taking the address of an argument, such as &argv1, causes DEC C for OpenVMS Alpha systems to generate prologue code for the function that moves all the arguments onto the stack (called **homing** arguments), causing a performance degradation. Also, argument list "walking" can be accomplished only by using the functions in the `<varargs.h>` or `<stdargs.h>` include files.

12.2.4.3 Synchronizing Exceptions

Because the Alpha architecture does not provide for immediate reporting of arithmetic exceptions, do not expect an assignment to a static variable (even with the **volatile** attribute) to occur before a subsequent exception is signaled.

12.2.4.4 Dynamic Condition Handlers

Although DEC C and DEC C++ for OpenVMS Alpha systems treat LIB$ESTABLISH as a built-in function, using LIB$ESTABLISH is not recommended on OpenVMS VAX or OpenVMS Alpha systems. C and C++ programmers should call VAXC$ESTABLISH instead of LIB$ESTABLISH (VAXC$ESTABLISH is a built-in function on DEC C and DEC C++ for OpenVMS Alpha systems).

12.2.5 SYS$STARLET_C.TLB: Functionally Equivalent to STARLETSD.TLB

OpenVMS Alpha Version 1.0 included a new file, SYS$STARLET_C.TLB, that contained all the .H files that provide STARLET functionality equivalent to STARLETSD.TLB. The file SYS$STARLET_C.TLB, together with DECC$RTLDEF.TLB now shipping with the DEC C Compiler, replaces VAXCDEF.TLB that previously shipped with the VAX C Compiler. DECC$RTLDEF.TLB contains all the .H files that support the compiler and RTL, such as STDIO.H.

The following differences may require source changes:

- RMS structures

  Previously, the RMS structures FAB, NAM, RAB, XABALL, and so forth, were defined in the appropriate .H files as "struct RAB {...", for example. The .H files supplied in OpenVMS Alpha Version 1.0 defined them as "struct rabdef {...". To compensate for this difference, lines of the form "#define RAB rabdef" were added. However, there is one situation where a source change is required because of this change. If you have a private structure that contains a pointer to one of these structures and your private structure is defined (but not used) before the RMS structure has been defined, you will receive compile-time errors similar to the following:

  ```
  %CC-E-PASNOTMEM, In this statement, "rab$b_rac" is not a member of "rab".
  ```

  This error can be avoided by reordering your source file so that the RMS structure is defined before the private structure. Typically, this involves moving around "#include" statements.

- LIB (privileged interface) structures

  Historically, three structures from LIB (NFBDEF.H, FATDEF.H, and FCHDEF.H) have been made available as .H files. These files were shipped as .H files in OpenVMS Alpha Version 1.0 and 1.5 (not in the new SYS$STARLET_C.TLB). In OpenVMS Alpha 7.0, the file SYS$LIB_C.TLB, containing all LIB structures and definitions, has been added. These three .H files are now part of that .TLB and are no longer shipped separately. Source changes may be required, as no attempt has been made to preserve any existing anomalies in these files. The structures and definitions from LIB are for privileged interfaces only and are therefore subject to change.

- Use of "variant_struct" and "variant_union"

  In the new .H files, "variant_struct" and "variant_union" are always used, whereas previously some structures used "struct" and "union". Therefore, the intermediate structure names cannot be specified when referencing fields within data structures.

For example, the following statement:

```
AlignFaultItem.PC[0] = DataPtr->afr$r_pc_data_overlay.afr$q_fault_pc[0];
```

becomes:

```
AlignFaultItem.PC[0] = DataPtr->afr$q_fault_pc[0];
```

- Member alignment

  Each of the .H files in SYS$STARLET_C.TLB saves and restores the state of "#pragma member_alignment".

- Conventions

  The .H files in SYS$STARLET_C.TLB adhere to some conventions that were only partly followed in VAXCDEF.TLB. All constants (#defines) have uppercase names. All identifiers (routines, structure members, and so forth) have lowercase names. Where there is a difference from VAXCDEF.TLB, the old symbol name is also included for compatibility, but users are encouraged to follow the new conventions.

- Use of Librarian utility to access the .H files

  During installation of OpenVMS Alpha, the contents of SYS$STARLET_C.TLB are not extracted into the separate .H files. The DEC C Compiler accesses these files from within SYS$STARLET_C.TLB, regardless of the format of the #include statement. If you want to inspect an individual .H file, you can use the Librarian utility, as in the following example:

```
$ LIBRARY /EXTRACT=AFRDEF /OUTPUT=AFRDEF.H SYS$LIBRARY:SYS$STARLET_C.TLB
```

- Additional .H files included in SYS$STARLET_C.TLB

  In addition to the .H files derived from STARLET sources, SYS$STARLET_C.TLB includes .H files that provide support for DECthreads, such as CMA.H.

12.2.6 VAX C Features Not Supported by /STANDARD=VAXC Mode

While most programming practices supported by VAX C are supported by DEC C for OpenVMS Alpha systems in /STANDARD=VAXC mode, certain programming practices that conflict with the ANSI standard are not supported. The following list highlights some of these differences; see the DEC C compiler documentation for more information.

- The inclusion of text after an #endif statement, as in the following example:

```
#ifdef a
   .
   .
   .
#endif a
```

  Delete the text or surround it with comment delimiters, as in the following:

```
#endif /* a */
```

- Modification of string constants, while always a questionable programming practice, was accepted by VAX C. DEC C for OpenVMS Alpha systems places all string constants in a read-only program section so that they cannot be modified.

- Structure-initialization values must be enclosed within braces ({}):

```
array[SIZE] = NULL;   /* accepted by VAX C */
array[SIZE] = {NULL}; /* required by DEC C */
```

- Redefinitions of symbols are now flagged with a warning-level diagnostic message:

```
#define x  a
#define x  b   /* generates a warning message in DEC C */
```

- Use of text libraries is no longer recommended. While supported by VAX C, text libraries are not portable.

```
#include  stdio
```

  Instead, use the following syntax:

```
#include <stdio.h>
```

- You must have one, and only one, declaration of an external variable. This is the definition of this variable. Other modules can use it by declaring it with the `extern` semantics.
- If you are recompiling VAX C code, either an entire application or one or more modules, you will want to pay particular attention to any external symbols that it contains. Unlike the VAX C compiler which supports one external symbol model, the DEC C compiler supports four models. The default external symbol produced by the DEC C compiler is not the same as the single VAX C external symbol.

  Furthermore, when you link such code, due to changes in the linker, if you did not specify the /SHARE qualifier when you recompiled the C code module, you will need to specify a related linker qualifier.

12.3 Compatibility of DEC COBOL with VAX COBOL

The DEC COBOL Version 1.0 compiler, running on an OpenVMS Alpha system, is based on and is highly compatible with the VAX COBOL Version 4.4 compiler running on an OpenVMS VAX system. The DEC COBOL compiler supports many, but not all, VAX COBOL features. The following list summarizes the major differences betweeen the DEC COBOL and VAX COBOL compilers:

- A new alignment qualifier that selects Alpha data alignment to optimize performance or VAX COBOL data alignment to ensure compatibility with VAX COBOL record alignment
- A new qualifier that provides both IEEE and VAX floating-point data types for single- and double-precision data items
- A new qualifier to generate code that allows native images to call translated images and translated images to call native images
- A new qualifier to recognize additional COBOL reserved words defined by the *X/Open Portability Guide*
- A new screen manager for ACCEPT/DISPLAY extensions
- Support for only the most important features of the VAX COBOL /STANDARD=V3 qualifier option
- No support for the VAX DBMS (Database Management System) Data Manipulation Language (DML)

- No support for intrinsic functions, which are included in VAX COBOL Version 5.0 and higher
- No support for multibyte characters and other Japanese-language features, which are included in Version 5.0 and higher of VAX COBOL (Japanese version)
- Support for file status values that are compatible with VAX COBOL Version 5.1, which differ from those of VAX COBOL Version 5.0 and previous versions

The information in this section is intended to help you write COBOL applications that are compatible with both VAX COBOL and DEC COBOL as well as to help you convert your existing COBOL applications from VAX COBOL to DEC COBOL.

This section describes similarities and differences between VAX COBOL Version 4.4 and DEC COBOL Version 1.0. Differences between DEC COBOL and later versions of VAX COBOL are noted when warranted.

For the latest information about product features and future release enhancements of the DEC COBOL compiler, refer to the most recent version of the DEC COBOL release notes. For information about VAX COBOL features, refer to the VAX COBOL release notes and other documentation. You can obtain an online version of the release notes for your installed COBOL compiler by entering the HELP COBOL RELEASE_NOTES command at the system prompt.

For reference information about DEC COBOL language features, see the *DEC COBOL Reference Manual*. For reference information about VAX COBOL language features, see the *VAX COBOL Reference Manual*. For information about DEC COBOL command line qualifiers, invoke the online help system for COBOL at the operating system prompt. For information about VAX COBOL command line qualifiers, see the *VAX COBOL User Manual*.

12.3.1 Command Line Qualifiers

Tables 12–6, 12–7, and 12–8 compare and contrast the DEC COBOL and VAX COBOL command line qualifiers.

12.3.1.1 Qualifiers Shared by DEC COBOL and VAX COBOL

Table 12–6 lists the command line qualifiers shared by DEC COBOL and VAX COBOL. For more information about the command line qualifiers available in DEC COBOL, refer to Table 12–7 or invoke the DEC COBOL online help system. For more information about the VAX COBOL command line qualifiers, refer to Table 12–8 and the *VAX COBOL User Manual*.

Table 12–6 Qualifiers and Options Shared by DEC COBOL and VAX COBOL

Qualifier	Comments
/ANALYSIS_DATA	Equivalent.
/ANSI_FORMAT	Equivalent.
/AUDIT	Equivalent.
/CHECK	A new option (/CHECK=[NO]DECIMAL) is available for DEC COBOL. (See Table 12–7 and Section 12.3.2.2.)

(continued on next page)

Table 12–6 (Cont.) Qualifiers and Options Shared by DEC COBOL and VAX COBOL

Qualifier	Comments
/CONDITIONALS	Equivalent.
/COPY_LIST	Equivalent.
/CROSS_REFERENCE	Equivalent.
/DEBUG	Equivalent.
/DEPENDENCY_DATA	Equivalent.
/DIAGNOSTICS	Equivalent.
/FIPS	Minor differences in functionality exist. (Invoke the DEC COBOL online help system for information about the behavior of the /FIPS=74 qualifier option.)
/FLAGGER	Equivalent.
/LIST	Equivalent.
/MACHINE_CODE	Equivalent.
/MAP	Equivalent.
/OBJECT	Equivalent.
/SEQUENCE_CHECK	Equivalent.
/STANDARD	Some VAX COBOL options are available in DEC COBOL. (See Section 12.3.2.7 for information about the behavior of the /STANDARD=V3 qualifier option.)
/TRUNCATE	Equivalent.
/WARNINGS	Minor differences in functionality exist. (See Section 12.3.2.7.2 and invoke the DEC COBOL online help system for information about the behavior of the /WARNINGS qualifier.)

12.3.1.2 DEC COBOL Qualifiers Not Available in VAX COBOL

Table 12–7 lists the command line qualifiers and options that are specific to DEC COBOL. These qualifiers and options are not available in VAX COBOL. For more information about the command line qualifiers available in DEC COBOL, invoke the DEC COBOL online help system.

Table 12–7 DEC COBOL Qualifiers Not Available in VAX COBOL

Qualifier	Comments
/ALIGNMENT=([NO]PADDING)	Aligns and pads data fields to conform with the Digital Calling Standards for OpenVMS Alpha and Digital UNIX.. (See Section 12.3.2.1.)
/CHECK=[NO]DECIMAL	Validates numeric digits when using display numeric items in a numeric context. (See Section 12.3.2.2.)
/CONVERT=LEADING_BLANKS	Changes leading blanks to zeros in numeric display items. (See Section 12.3.2.3.)

(continued on next page)

Table 12–7 (Cont.) DEC COBOL Qualifiers Not Available in VAX COBOL

Qualifier	Comments
/FLOAT=[D_FLOAT],[IEEE_FLOAT]	Specifies the floating-point data format to be used in memory for single- and double-precision data items. (See Section 12.3.2.4.)
/OPTIMIZE	Controls whether the compiler optimizes the compiled program to generate more efficient code. (See Section 12.3.2.5.)
/RESERVED_WORDS=([NO]XOPEN, [NO]FOREIGN_EXTENSIONS)	Controls whether or not the compiler recognizes X/Open COBOL words as reserved words. (See Section 12.3.2.6.)
/TIE	Generates code that allows native images to call translated images and translated images to call native images. (See Section 12.3.2.8.)

12.3.1.3 VAX COBOL Qualifiers Not Available in DEC COBOL

Table 12–8 lists the command line qualifiers and options that are specific to VAX COBOL. These qualifiers and options are not available in DEC COBOL. For detailed information about the VAX COBOL command line qualifiers, refer to the *VAX COBOL User Manual*.

Table 12–8 VAX COBOL Qualifiers Not Available in DEC COBOL

Qualifier	Comments
/DESIGN	Controls whether the compiler processes the input file as a detailed design.
/INSTRUCTION_SET[=option]	Improves run-time performance on single-chip VAX processors, using different portions of the VAX instruction set.
/STANDARD=[NO]OPENVMS_ Alpha	Produces informational messages about language features that are not supported by the DEC COBOL compiler. (See Section 12.3.2.9 and the VAX COBOL Version 5.1 release notes.)
/STANDARD=[NO]PDP11	Produces informational messages about language features that are not supported by the COBOL–81 compiler.
/WARNINGS=[NO]STANDARD	Produces informational messages about language features that are Digital extensions. The DEC COBOL equivalent is /STANDARD=[NO]SYNTAX. (See Section 12.3.2.7.2.)

12.3.2 Behavior Differences

This section describes differences in behavior between VAX COBOL Version 4.4 and DEC COBOL Version 1.0, including new DEC COBOL command line qualifiers and options, as well as behavior that is specific to DEC COBOL Version 1.0.

12.3.2.1 Specifying Alignment for Numeric Data Items with the DEC COBOL /ALIGNMENT Qualifier and Alignment Directives

You can use the /ALIGNMENT qualifier and alignment directives to specify the alignment of binary and decimal data items within record structures. Refer to the *DEC COBOL Reference Manual* for specific information about alignment.

Proper data alignment is needed to optimize your COBOL applications on Alpha systems. Manipulating binary data is significantly faster if the data lies within natural boundaries. Manipulating decimal data is significantly faster if you align the data along the preferred boundaries for the system you are using.

The primary goal of alignment specification is optimum performance. In addition, the /ALIGNMENT qualifier and alignment directives meet the following goals:

- Ease of use and conversion—You need to make only a minimal number of changes to your existing source files. In some cases, all you need to do is add the /ALIGNMENT qualifier when you invoke the DEC COBOL compiler.
- VAX COBOL source compatibility—You can compile the same source files with VAX COBOL and DEC COBOL. DEC COBOL directives are structured comments that the VAX COBOL compiler ignores.
- Flexibility—You can specify VAX byte alignment or natural alignment on a record-by-record basis. For example, you can specify byte alignment for files shared by both compilers and natural alignment for DEC COBOL files and records.

The /ALIGNMENT qualifier, alignment directives, and SYNCHRONIZED clause affect the alignment of items within a group (group elements) as shown in the following figure:

```
                    /ALIGNMENT qualifier

Alignment directive                          SYNCHRONIZED clause



                    Align binary and decimal  Align binary
                    group elements            group elements
```

As with the VAX COBOL compiler, you can use the SYNCHRONIZED clause to align binary components of records on natural boundaries. Thus, for the DEC COBOL compiler operating on binary data, the SYNCHRONIZED clause, /ALIGNMENT qualifier, and alignment directives can exhibit equivalent behavior.

12.3.2.1.1 Using the /ALIGNMENT Qualifier The /ALIGNMENT qualifier allows you to specify natural alignment for binary data and preferred alignment for numeric decimal data in your program.

Binary and decimal alignment are separate options (a useful feature for programs that alias decimal and string data, but that can still benefit from the alignment of binary data). For example, when you specify /ALIGNMENT=(BINARY,NODECIMAL) (or /ALIGNMENT), the DEC COBOL compiler aligns binary data along natural boundaries and decimal data along byte boundaries. Use /ALIGNMENT to ensure that your data is aligned for optimum performance on OpenVMS Alpha systems.

Use /NOALIGNMENT, the default, to specify byte data alignment (including programs that align binary data items with the SYNCHRONIZED clause). Also use /NOALIGNMENT for portability and compatibility with data files produced on an OpenVMS VAX system.

12.3.2.1.2 Using Alignment Directives The alignment properties specified by the /ALIGNMENT qualifier remain in effect throughout a given compilation, except as modified by alignment directives. **Directives** are structured comments that the DEC COBOL compiler interprets. (DEC COBOL directives are ignored by the VAX COBOL compiler.) All directives begin with "*DC", where the "*" signals the beginning of the structured comment.

You can use the following alignment directives anywhere within your COBOL source program to change the current set of alignment porperties:

- *DC SET ALIGNMENT[=(option,...)] (where *option* is [NO]BINARY or [NO]DECIMAL)—Specifies a new alignment. Specifying *DC SET ALIGNMENT is equivalent to specifying *DC SET ALIGNMENT=(BINARY,NODECIMAL).
- *DC END-SET ALIGNMENT—Restores the alignment to the previous setting. (Use of this alignment directive is optional.)
- *DC SET NOALIGNMENT—Specifies byte alignment.

You can nest alignment directives within your program to turn alignment on or off for specific numeric data items. Although the *DC END-SET ALIGNMENT directive is optional, you must use it to indicate the end of each nested alignment directive.

12.3.2.2 Validating Numeric Data with the DEC COBOL /CHECK=NODECIMAL Qualifier Option

The /CHECK=[NO]DECIMAL qualifier option validates numeric characters when you use numeric display items in a numeric context. Use /CHECK=DECIMAL when you want the system to generate an error for any invalid, or nonnumeric, characters.

This feature is primarily intended to help validate data produced by other systems that might use a different internal representation for numeric data. A secondary consideration is that this qualifier can also be used to detect logic errors in programs that result in text data being moved to numeric data items. The disadvantage of this feature is that extra instructions are needed to perform the checks, resulting in slightly larger images and slightly longer execution times.

Use /CHECK=NODECIMAL, the default, when you do not want the system to check for numeric characters in numeric display items.

12.3.2.3 Converting Leading Blanks to Zeros with the DEC COBOL /CONVERT=LEADING_BLANKS Qualifier Option

The /CONVERT=LEADING_BLANKS qualifier and option generates code to check for and change leading blanks to zeros in numeric display items.

This feature is primarily intended to help users convert existing COBOL programs to run on an OpenVMS Alpha system by changing leading blanks in the data to zeros at run time. The disadvantage of this feature is that extra instructions are needed to perform the data conversions. This results in slightly larger images and slightly longer execution times.

Use /NOCONVERT=LEADING_BLANKS, the default, when you do not want the compiler to change leading blanks to zeros in numeric display items.

12.3.2.4 Specifying a Floating-Point Data Format with the DEC COBOL /FLOAT Qualifier

The /FLOAT=[option] qualifier specifies the floating-point data format to be used in memory for single- and double-precision data items. Specify either /FLOAT=D_FLOAT or /FLOAT=IEEE_FLOAT within a single program.

Because the Alpha architecture is IEEE-compliant, you can run existing COBOL programs containing IEEE floating-point data formats on DEC COBOL.

Use the /FLOAT=D_FLOAT qualifier option, the default, at compile time to specify the VAX F_floating memory format for single-precision (COMP-1) data and the VAX D_floating memory format for double-precision (COMP-2) data.

The IEEE standard for binary floating-point arithmetic, ANSI/IEEE 754-1985, defines four floating-point formats in two groups, basic and extended, each group having two widths, single and double. The Alpha architecture supports the basic single and double formats.

Use the /FLOAT=IEEE_FLOAT qualifier option at compile time to specify the IEEE S_floating memory format for single-precision (COMP-1) data and the IEEE T_floating memory format for double-precision (COMP-2) data.

Refer to the *Alpha Architecture Handbook* for more information about using floating-point data types with the Alpha architecture.

12.3.2.5 Optimizing Your Code with the DEC COBOL /OPTIMIZE Qualifier

The /OPTIMIZE qualifier controls whether the compiler optimizes the compiled program to generate more efficient code.

Use /OPTIMIZE, the default, when you want your program to run faster. Note that using this qualifier may cause the compiler to produce larger object modules and result in longer compile times.

Use /NOOPTIMIZE for a debugging session to ensure that the machine code occurs in the same order as the program lines in your source program.

12.3.2.6 Checking for Special Reserved Words with the DEC COBOL /RESERVED_WORDS Qualifier

The /RESERVED_WORDS qualifier controls whether or not the compiler recognizes certain COBOL words as reserved words.

Use /RESERVED_WORDS=NOXOPEN if your program uses one or more of the COBOL words defined by the *X/Open Portability Guide* as an identifier.

Use /RESERVED_WORDS=XOPEN, the default, if none of the following X/Open COBOL words appears in your program:

AUTO
BACKGROUND-COLOR
BELL
BLINK
EOL
EOS
ERASE
FOREGROUND-COLOR
FULL
HIGHLIGHT
LOWLIGHT
REQUIRED
REVERSE-VIDEO
SCREEN

SECURE
UNDERLINE

12.3.2.7 Calling Out Language Feature Extensions to the COBOL ANSI Standard with the DEC COBOL /STANDARD Qualifier

The /STANDARD qualifier controls whether the compiler prints informational messages associated with specific language features. To receive these messages, specify /STANDARD or /STANDARD=85 (and /WARNINGS=ALL or /WARNINGS=INFORMATIONAL) or /STANDARD=SYNTAX.

Use /STANDARD=85, the default, to instruct the DEC COBOL compiler to compile and generate code according to the ANSI 1985 COBOL standard.

Use /STANDARD=SYNTAX to instruct the DEC COBOL compiler to produce informational messages about language features that are Digital extensions to the ANSI 1985 COBOL Standard. The default, NOSYNTAX, suppresses these messages.

Use /STANDARD=V3 to instruct the DEC COBOL compiler to compile and generate code in the manner of VAX COBOL Version 3.4 in specific instances. Section 12.3.2.7.1 describes the /STANDARD=V3 qualifier option in more detail.

12.3.2.7.1 /STANDARD=V3 Qualifier Option DEC COBOL Version 1.0, as with VAX COBOL Version 4.0 and higher versions, is based on the ANSI 1985 COBOL standard. As such, DEC COBOL provides full support for the /STANDARD=85 qualifier option. DEC COBOL also provides support for some features of the /STANDARD=V3 qualifier option that were available with VAX COBOL Version 4.0 and higher.

VAX COBOL versions prior to Version 4.0 were based on the ANSI 1974 COBOL standard. While most of the enhancements made to VAX COBOL Version 4.0 and higher versions are compatible with earlier versions of the VAX COBOL compiler, some differences exist, which cause results to vary in some instances.

To minimize conflicts with existing VAX COBOL programs, VAX COBOL allows you to compile programs according to the rules for either VAX COBOL Version 4.0 and later versions or VAX COBOL Version 3.4. Specifying /STANDARD=V3 instructs the VAX COBOL compiler to compile and generate code in the manner of VAX COBOL Version 3.4 in specific instances, as described in the *VAX COBOL User Manual*.

When compared with the features available with VAX COBOL Version 4.0 and higher, DEC COBOL provides limited support for the /STANDARD=V3 qualifier option. When you specify /STANDARD=V3, DEC COBOL behavior is identical to VAX COBOL Version 4.0 and higher behavior in the following four specific instances:

- EXIT PROGRAM statement in a main program
- I/O file status codes
- No valid next record condition
- Opening nonoptional files in I/O and EXTEND mode

The following four subsections describe this DEC COBOL behavior in more detail.

EXIT PROGRAM Statement

If you specify /STANDARD=V3, an EXIT PROGRAM statement is treated as a return in both main programs and subprograms.

Specifying /STANDARD=85 bypasses an EXIT PROGRAM statement in the body of a main program and executes the statements following the EXIT PROGRAM statement. If the program is a subprogram, the EXIT PROGRAM statement acts as a return to the program that called the subprogram.

I/O File Status Codes

If you specify /STANDARD=V3, you receive the file status codes listed in the left-hand column, labeled V3, and your program acts accordingly.

If you specify /STANDARD=85, you receive the file status codes listed in the right-hand column, labeled 85, and your program acts accordingly.

Table 12–9 explains the I/O file status codes for VAX COBOL Version 3.4 and DEC COBOL.

Table 12–9 I/O File Status Codes for the /STANDARD Qualifier

	Status Code	
I/O Error Condition	V3	85
READ successful—record shorter than fixed file attribute.	00	04
CLOSE reel/unit attempted on nonreel/unit device.	00	07
READ fails—relative key digits exceed relative key.	00	14
WRITE fails—relative key digits exceed relative key.	00	24
OPEN I/O on file that is not mass storage.	00	37
WRITE fails—attempt to write a record of a different size than in the file description.	00	44
READ fails—no next logical record (EOF detected).	13	10
READ fails—no next logical record (EOF on OPTIONAL file).	15	10
READ fails—no valid next record (already at EOF).	16	10
READ NEXT or sequential READ—no valid next record pointer.	16[1]	46[1]
READ or START fails—optional input file not present.	25	23
READ successful—record longer than fixed file attribute.	30	04
OPEN on relative or indexed file that is not mass storage.	30	37
REWRITE fails—attempt to rewrite record of different size.	30	44
CLOSE fails—file not currently open.	93	42
DELETE or REWRITE fails—previous I/O not successful READ.	93	43
OPEN fails—file previously closed with LOCK.	94	38
OPEN fails—file created with different organization.	94	39
OPEN fails—file created with different prime record key.	94	39
OPEN fails—file created with different alternate record keys.	94	39

[1]See the subsection No Valid Next Record Condition.

(continued on next page)

Table 12–9 (Cont.) I/O File Status Codes for the /STANDARD Qualifier

	Status Code	
I/O Error Condition	**V3**	**85**
OPEN fails—file currently open.	94	41
READ or START fails—file not opened INPUT or I/O.	94	47
WRITE fails—file not opened OUTPUT, EXTEND, or I/O.	94	48
DELETE or REWRITE fails—file not opened I/O.	94	49
OPEN INPUT on a nonoptional file—file not found.	97	35

No Valid Next Record Condition

This subsection describes what happens when you compile your program using either /STANDARD=V3 or /STANDARD=85 and when all the following conditions exist:

- The no valid next record (NVNR) condition exists.
- Your program attempts a sequential READ statement.
- Your program includes an AT END branch associated with the READ statement.

When you use /STANDARD=V3 to compile your program, the following occurs:

- The file status code variable, if any, for the file is set to 16.
- The statements associated with the AT END statement are executed.
- The program continues to execute normally.

If you use /STANDARD=85 to compile your program, the following occurs:

- The file status code variable, if any, for the file is set to 46.
- The statements associated with the AT END statement are not executed.
- The program terminates execution abnormally (unless you have provided for this situation with a USE AFTER STANDARD EXCEPTION procedure).

OPEN I/O and EXTEND Modes

If you specify /STANDARD=V3, nonoptional files opened in I/O or EXTEND mode are created, if the files are unavailable.

If you specify /STANDARD=85, nonoptional files opened in I/O or EXTEND mode are not created if the files are unavailable. Instead, a run-time error is issued.

12.3.2.7.2 /STANDARD and /WARNINGS Qualifiers VAX COBOL provides two qualifiers that specify the same behavior: /STANDARD=[NO]SYNTAX and /WARNINGS=[NO]STANDARD.

DEC COBOL does not support the [NO]STANDARD option of the /WARNINGS qualifier. Therefore, specifying /WARNINGS=ALL with the DEC COBOL compiler will not produce the informational messages that point out Digital extensions. To receive messages such as the following one, you must specify /STANDARD=SYNTAX.

```
%COBOL-I-EXTENSION
```

Note

For VAX COBOL and DEC COBOL, the FIPS messages about Digital extensions that the compiler produces when you specify /FLAGGER[(=option, . . .)] continue to be controlled by the /WARNINGS=INFORMATION qualifier option.

12.3.2.8 Calling Native and Translated Images with the DEC COBOL /TIE Qualifier

The /TIE (Translated Image Environment) qualifier generates code that allows native OpenVMS Alpha images to call translated images and translated images to call native OpenVMS Alpha images. This qualifier is supported on OpenVMS Alpha systems only.

Specifying /TIE enables you to use compiled code with shared translated images, either because the code might call into a translated image or because it might be called from a translated image. If you specify /TIE, you should link the object module using the LINK command qualifier /NONATIVE_ONLY. (See the *OpenVMS Linker Utility Manual* for information about the /NONATIVE_ONLY qualifier.)

Specifying /NOTIE, the default, indicates that your compiled code will not be associated with a translated image.

For information about interoperability, see Chapter 11. For information about translated images, see *DECmigrate for OpenVMS AXP Systems Translating Images*.

12.3.2.9 VAX COBOL to DEC COBOL Program Conversion

VAX COBOL Version 5.1 provides a new flagging system, via the /STANDARD=OPENVMS_Alpha qualifier option, to identify language features in your existing VAX COBOL programs that are not available in DEC COBOL on OpenVMS Alpha.

When you specify /STANDARD=OPENVMS_Alpha (and /WARNINGS=ALL or /WARNINGS=INFORMATIONAL), the VAX COBOL compiler generates informational messages to flag language constructs that are not available in DEC COBOL. You can use this information to modify your program before running it on DEC COBOL.

Use /STANDARD=NOOPENVMS_Alpha, the default, to suppress these informational messages.

12.3.2.10 Program Structure

In some cases, the DEC COBOL compiler generates more complete messages about unreachable code or other logic errors than does the VAX COBOL compiler.

The following example shows a sample program and the messages issued by the DEC COBOL compiler.

Source file:

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. T1.
       ENVIRONMENT DIVISION.
       PROCEDURE DIVISION.
       P0.
           GO TO P1.
       P3.
           GO TO P2.
       P2.
           DISPLAY "This is unreachable code".
       P1.
           STOP RUN.
       IDENTIFICATION DIVISION.
       PROGRAM-ID. T2.
       ENVIRONMENT DIVISION.
       PROCEDURE DIVISION.
       P0.
           DISPLAY "This is unreachable code".
           EXIT PROGRAM.
       END PROGRAM T2.
       END PROGRAM T1.
```

On VAX systems:

```
$ COBOL /ANSI/WARNINGS=ALL T1.COB
```

On Alpha systems:

```
$ COBOL/ANSI/OPT/WARNINGS=ALL T1.COB
       PROGRAM-ID. T2.
..................^
%COBOL-I-UNCALLED, routine T2 can never be called
at line number 14 in file DISK$YOURDISK:[TESTDIR]T1.COB;1
       P2.
.......^
%COBOL-I-UNREACH, code can never be executed at label P2
at line number 9 in file DISK$YOURDISK:[TESTDIR]T1.COB;1
```

For the same program, the VAX COBOL compiler produces no messages even though the compiler does detect both the unreachable label *and* the unreachable contained program.

Use the /OPTIMIZE qualifier to direct the DEC COBOL compiler to do the uncalled routine analysis. The compiler performs the unreachable code analysis for the default (lowest) level of optimization.

This difference from VAX COBOL can help you when debugging a program. Because these messages are informational, the compiler produces an object file, which you can link and execute. However, these messages may point out otherwise undetected logic errors (that is, the structure of the program is probably not what you intended).

12.3.2.11 COPY and REPLACE Statements

The DEC COBOL compiler produces different output when listing annotations for the COPY statement in COBOL programs.

The following two examples show the difference in the position of the listing annotations, represented by the letter L, in a COBOL program using the VAX COBOL compiler and the DEC COBOL compiler.

VAX COBOL source file:

```
    1        IDENTIFICATION DIVISION.
    2        PROGRAM-ID. DCOP1B.
    3       *
    4       *    This program tests the copy library file
    5       *    with a comment in the middle of it.
    6       *    It should not produce any diagnostics.
    7            COPY
    8       *    This is the comment in the middle
    9                    LCOP1A.
   10L       ENVIRONMENT DIVISION.
   11L       INPUT-OUTPUT SECTION.
   12L       FILE-CONTROL.
   13L       SELECT FILE-1
   14L           ASSIGN TO "FILE1.TMP".
   15        DATA DIVISION.
   16        FILE SECTION.
   17        FD  FILE-1.
   18        01  FILE1-REC       PIC X.
   19        WORKING-STORAGE SECTION.
   20        PROCEDURE DIVISION.
   21        PE. DISPLAY "***END***"
   22            STOP RUN.
```

DEC COBOL source file:

```
             1 IDENTIFICATION DIVISION.
             2 PROGRAM-ID. DCOP1B.
             3 *
             4 *       This program tests the copy library file.
             5 *       with a comment in the middle of it.
             6 *       It should not produce any diagnostics.
             7         COPY
             8 *       this is the comment in the middle
             9                 LCOP1A.
L           10 ENVIRONMENT DIVISION.
L           11 INPUT-OUTPUT SECTION.
L           12 FILE-CONTROL.
L           13 SELECT FILE-1
L           14         ASSIGN TO "FILE1.TMP".
            15 DATA DIVISION.
            16 FILE SECTION.
            17 FD      FILE-1.
            18 01      FILE1-REC       PIC X.
            19 WORKING-STORAGE SECTION.
            20 PROCEDURE DIVISION.
            21 PE.     DISPLAY "***END***"
            22         STOP RUN.
```

The DEC COBOL compiler also produces different output when listing a COBOL program with multiple COPY statements on a single line, as shown in the next two examples. When the compiler issues a message on a replaced line, the message pointer calls out the original text, not the replacement text.

VAX COBOL source file:

```
     1        IDENTIFICATION DIVISION.
     2        PROGRAM-ID. DCOP1J.
     3       *
     4       *    Tests copy with three copy statements on 1 line.
     5       *
     6        ENVIRONMENT DIVISION.
     7        DATA DIVISION.
     8        PROCEDURE DIVISION.
     9        THE.
    10            COPY LCOP1J.
    11L           DISPLAY "POIUYTREWQ".
    12C                        COPY LCOP1J.
    13L           DISPLAY "POIUYTREWQ".
    14C                                     COPY LCOP1J.
    15L           DISPLAY "POIUYTREWQ".
    16            STOP RUN.
```

DEC COBOL source file:

```
              1 IDENTIFICATION DIVISION.
              2 PROGRAM-ID. DCOP1J.
              3 *
              4 *       Tests copy with three copy statements on 1 line.
              5 *
              6 ENVIRONMENT DIVISION.
              7 DATA DIVISION.
              8 PROCEDURE DIVISION.
              9 THE.
             10         COPY LCOP1J. COPY LCOP1J. COPY LCOP1J.
L            11         DISPLAY "POIUYTREWQ".
L            12         DISPLAY "POIUYTREWQ".
L            13         DISPLAY "POIUYTREWQ".
             14         STOP RUN.
```

The diagnostics for the COBOL source statements REPLACE and DATE-COMPILED result in compiler listings that contain multiple instances of the source line.

For a REPLACE statement listing in a DEC COBOL program, if the compiler issues a message on the replacement text, the compiler message corresponds to the original text in the program. In a VAX COBOL program, however, the compiler message corresponds to the replacement text.

The compiler listing for a DEC COBOL program and a VAX COBOL program differs when a COPY statement inserts text in the middle of a line as shown in the following two examples.

DEC COBOL source file:

```
-----------------------------------------------------------
   13 P0.     MOVE COPY LCOP5D. TO ALPHA.
L  14                 "O"
```

VAX COBOL source file:

```
-----------------------------------------------------------
13        P0. MOVE COPY LCOP5D.
14L               "O"
15C                               TO ALPHA.
```

LCOP5D.LIB contains "O". The DEC COBOL compiler keeps the same line and inserts the COPY file contents below the source line. The VAX COBOL compiler splits the original source line into parts.

For the REPLACE and COPY REPLACING statements, program listing line numbers differ between DEC COBOL and VAX COBOL. For DEC COBOL, the line number for the replacement line corresponds to its line number in the original source text, while subsequent line numbers differ. The VAX COBOL compiler arranges the line numbers consecutively.

The following source program can result in listings with different ending line numbers, depending on whether you compile it with the DEC COBOL or the VAX COBOL compiler.

Source file:

```
REPLACE ==A VERY LONG STATEMENT== by ==EXIT PROGRAM==.
A
VERY
LONG
STATEMENT.
DISPLAY "To REPLACE or not to REPLACE".
```

DEC COBOL version:

```
------------------------------------------------------------------
        1 REPLACE ==A VERY LONG STATEMENT== by ==EXIT PROGRAM==.
        2 EXIT PROGRAM.
        6 DISPLAY "To REPLACE or not to REPLACE".
```

VAX COBOL version:

```
------------------------------------------------------------------
        1 REPLACE ==A VERY LONG STATEMENT== by ==EXIT PROGRAM==.
        2 EXIT PROGRAM.
        3 DISPLAY "To REPLACE or not to REPLACE".
```

12.3.2.12 MOVE Statement

Unsigned computational fields can hold larger values than signed computational fields. In accordance with the ANSI COBOL Standard, the values for unsigned items should always be treated as positive. VAX COBOL, however, treats unsigned items as signed, while DEC COBOL treats them as positive. Therefore, in some rare cases, a mixture of unsigned and signed data items in MOVE or arithmetic statements can produce different results between VAX COBOL and DEC COBOL.

The following sample program produces different results for VAX COBOL and DEC COBOL.

Source file:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SHOW-DIFF.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 A2        PIC 99    COMP.
01 B1        PIC S9(5) COMP.
01 B2        PIC 9(5)  COMP.
PROCEDURE DIVISION.
TEST-1.
    MOVE 65535 TO A2.
    MOVE A2 TO B1.
    DISPLAY B1 WITH CONVERSION.
    MOVE A2 TO B2.
    DISPLAY B2 WITH CONVERSION.
    STOP RUN.
```

VAX COBOL results:

```
B1 =  -1
B2 =  -1
```

DEC COBOL results:

```
B1 =  65535
B2 =  65535
```

12.3.2.13 ACCEPT and DISPLAY Statements

When you use any extended feature of ACCEPT or DISPLAY within your program, the DEC COBOL compiler uses the DEC SMG (Screen Manager). The visible differences in behavior between DEC COBOL and VAX COBOL are as follows:

- When you run your program, the screen is automatically erased when it encounters the first ACCEPT or DISPLAY statement.
- Because the DEC SMG manages terminal I/O use with extended ACCEPT and DISPLAY statements as screen entities rather than as line by line I/O, you may not be able to redisplay information that appears to have scrolled off the screen by using the DECterm scroll bar.
- The DCL RECALL command is not supported during screen accepts.
- Escape sequence processing is limited to the use of an escape sequence that occupies the leftmost positions of a DISPLAY string. (Sample programs are located in the *DEC COBOL User Manual.*)
- When you mix ANSI ACCEPT statements and extended ACCEPT statements in a program, the editing keys used by the extended ACCEPT statements will also be used by the ANSI ACCEPT statements. (See the *DEC COBOL User Manual* for a complete list of editing keys.)

12.3.2.14 LINAGE Statement

The DEC COBOL and VAX COBOL compilers exhibit different behavior when handling large values for the LINAGE statement. If the line count for the ADVANCING clause of the WRITE statement is larger than 127, DEC COBOL advances one line. VAX COBOL results are undefined.

12.3.2.15 File Status Differences

The DEC COBOL and VAX COBOL compilers report different file status codes when you open a file in EXTEND mode and then try to REWRITE it. DEC COBOL reports a 49 (incompatible open mode). VAX COBOL reports an error 43 (no previous READ).

DEC COBOL sets the file status to 46 after a START fails. VAX COBOL does not produce these results.

12.3.2.16 System Return Codes

The following example illustrates an illegal coding practice that exhibits a certain behavior on OpenVMS VAX systems but that does not produce the same behavior on OpenVMS Alpha systems. This difference in behavior points to an architectural difference in the register sets between the VAX and Alpha architectures. Specifically, the difference in behavior on the Alpha system is due to the separate set of registers used for floating-point data types.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. BADCODING.
ENVIRONMENT DIVISION.

DATA DIVISION.
FILE SECTION.

WORKING-STORAGE SECTION.

    01  FIELDS-NEEDED.
        05 CYCLE-LOGICAL      PIC X(14) VALUE 'A_LOGICAL_NAME'.

    01  EDIT-PARM.
        05  EDIT-YR            PIC X(4).
        05  EDIT-MO            PIC XX.

    01  CMR-RETURN-CODE        COMP-1 VALUE 0.

LINKAGE SECTION.

       01 PARM-REC.
          05 CYCLE-PARM        PIC X(6).
          05 RETURN-CODE       COMP-1 VALUE 0.

PROCEDURE DIVISION USING PARM-REC GIVING CMR-RETURN-CODE.

P0-CONTROL.

    CALL 'LIB$SYS_TRNLOG' USING BY DESCRIPTOR CYCLE-LOGICAL,
                                OMITTED,
                                BY DESCRIPTOR CYCLE-PARM
                                GIVING RETURN-CODE.

    IF  RETURN-CODE  GREATER 0
      THEN
        MOVE RETURN-CODE TO CMR-RETURN-CODE
        GO TO P0-EXIT.

    MOVE  CYCLE-PARM  TO  EDIT-PARM.

    IF  EDIT-YR  NOT  NUMERIC
      THEN
        MOVE  4  TO CMR-RETURN-CODE, RETURN-CODE.

    IF  EDIT-MO  NOT NUMERIC
      THEN
        MOVE  4  TO  CMR-RETURN-CODE, RETURN-CODE.
```

```
    IF  CMR-RETURN-CODE GREATER 0
                 OR
        RETURN-CODE GREATER 0
      THEN
        DISPLAY "****************************"
        DISPLAY "** BADCODING.COB  **"
        DISPLAY "** A_LOGICAL_NAME> ", CYCLE-PARM, "    **"
        DISPLAY "****************************".

P0-EXIT.

    EXIT PROGRAM.
```

In the sample program, the programmer incorrectly defined the return value for a system service call to be F_floating when it should have been binary (COMP). The programmer was depending on the following VAX behavior: in the VAX architecture, all return values from routines are returned in register R0. The VAX architecture has no separate integer and floating-point registers. The Alpha architecture defines separate register sets for floating-point and binary data. In particular, routines that return floating-point values return them in register F0; routines that return binary values return them in register R0.

The DEC COBOL compiler has no method for determining what data type an external routine may return. You must specify the correct data type for the GIVING-VALUE item in the CALL statement. On OpenVMS Alpha systems, the generated code is testing F0 instead of R0 because of the different set of registers used for floating-point data items.

In the sample program, the value in F0 is completely random in this code sequence. In some cases, this coding practice may produce the expected behavior, but in most cases it will not.

12.3.2.17 Storage Differences for Double-Precision Data Items

The difference in storage of D_floating items between the VAX and Alpha architectures produces slightly different answers when validating execution results. The magnitude of the difference depends upon how many D-float computations and stores the compiler performed before outputing the final answer. This behavior difference may cause some difficulty if you attempt to validate output generated by your program running on OpenVMS Alpha systems against output generated by OpenVMS VAX systems where they output COMP-2 data to a file.

For information about storage for floating-point data types, see the *Alpha Architecture Handbook*.

12.3.2.18 RMS Special Registers

The DEC COBOL run-time system checks some I/O error situations before attempting the RMS operation. VAX COBOL does the RMS calls without doing any checking, resulting in different values for RMS special registers. When the DEC COBOL run-time system does not attempt an RMS operation, the register value retains its previous value.

For example, in the case of a file that was not successfully opened, any DEC COBOL record operation (READ, WRITE, START, DELETE, REWRITE, or UNLOCK) will fail without invoking RMS.

12.4 Compatibility of Digital Fortran for OpenVMS Alpha with VAX FORTRAN

This section discusses the compatibility between Digital Fortran for OpenVMS Alpha systems and DEC Fortran for OpenVMS VAX Systems (formerly VAX FORTRAN) in the following areas:

- Language features (Section 12.4.1)
- Command line qualifiers (Section 12.4.2)
- Interoperability with translated shared images (Section 12.4.3)
- Porting DEC Fortran for OpenVMS VAX Systems data (Section 12.4.4)

12.4.1 Language Features

Digital Fortran for OpenVMS Alpha includes ANSI FORTRAN–77 and ISO/ANSI Fortran 9x standard features, as well as the Digital Fortran for OpenVMS VAX Systems extensions to these Fortran standards, including:

- RECORD statement and STRUCTURE statement
- CDEC$ directives and the OPTIONS statement
- BYTE, INTEGER*1, INTEGER*2, INTEGER*4, LOGICAL*1, LOGICAL*2, LOGICAL*4
- REAL*4, REAL*8, REAL*16, COMPLEX*8, COMPLEX*16
- IMPLICIT NONE statement
- INCLUDE statement
- NAMELIST I/O
- Names up to 31 characters including use of dollar sign ($) and underscore (_)
- DO WHILE and END DO statements
- Use of the exclamation point (!) for end-of-line comments
- Built-in functions %DESCR, %LOC, %REF, and %VAL
- VOLATILE statement
- DICTIONARY statement (FORTRAN–77 compiler only)
- POINTER statement data type
- Recursion
- Unformatted data conversion between disk and memory
- Indexed files
- I/O statements such as PRINT, ACCEPT, TYPE, DELETE, UNLOCK
- OPEN and INQUIRE statement specifiers, including CARRIAGECONTROL, CONVERT, ORGANIZATION, RECORDTYPE
- Other language elements identified in the appropriate Fortran language reference manuals

For detailed information about extensions and language features, see the Fortran langauge reference manual, which visually shows extensions of the FORTRAN–77 standard.

Note

The Digital Fortran for OpenVMS Alpha product supports most of the FORTRAN–77 language extensions supported by DEC Fortran for OpenVMS Alpha and the ISO/ANSI Fortran 90 standard.

The remainder of this section summarizes language features specific to DEC Fortran for OpenVMS VAX Systems and Digital Fortran for OpenVMS Alpha, language features that are shared but interpreted differently in each language, Digital Fortran for OpenVMS Alpha restrictions that do not apply to DEC Fortran for OpenVMS VAX Systems, and data porting considerations.

12.4.1.1 Language Features Specific to Digital Fortran for OpenVMS Alpha

The following language features are available in Digital Fortran for OpenVMS Alpha but are not supported in Digital Fortran for OpenVMS VAX Systems Version 6.3:

- Quotation marks (") as delimiters for character constants. This can be disabled by specifying the /VMS qualifier.
- Naturally aligned or packed boundaries for fields of records and items in COMMON blocks
- The INTEGER*1, INTEGER*8, and LOGICAL*8 data types
- Support for S_floating, T_floating, and X_floating IEEE data types as well as support for non-native unformatted data file formats, including big-endian numeric format. For a description of the native floating-point data types for Alpha systems, see the *Alpha Architecture Reference Manual*.
- LIB$ESTABLISH and LIB$REVERT are provided as intrinsic functions for compatibility with DEC Fortran for OpenVMS VAX Systems condition handling.

  DEC Fortran converts declarations to LIB$ESTABLISH to DEC Fortran RTL specific entry points.
- The alternate "Z" spelling for double-precision complex intrinsic functions. (For example, the square root double-precision intrinsic function can be spelled as CDSQRT or ZSQRT.)
- The following intrinsic functions:

  IMAG
  AND
  OR
  XOR
  LSHIFT
  RSHIFT
- Certain run-time errors are specific to Digital Fortran for OpenVMS Alpha.
- Case-sensitive names

- I/O unit numbers can be any nonnegative integer in Digital Fortran for OpenVMS Alpha. In DEC Fortran for OpenVMS VAX Systems, the values for I/O unit numbers can range from 0 to 99.

For an explanation of Digital Fortran language features, see the Fortran language reference manual.

12.4.1.2 Language Features Specific to DEC Fortran for OpenVMS VAX Systems

The following language features are available in DEC Fortran for OpenVMS VAX Systems but are not supported in Digital Fortran for OpenVMS Alpha:

- Automatic decomposition features of FORTRAN/PARALLEL=(AUTOMATIC)
- Manual (directed) decomposition features of FORTRAN /PARALLEL=(MANUAL) using the CPAR$ directives, such as CPAR$ DO_PARALLEL
- The following I/O and error subroutines for PDP–11 compatibility:

ASSIGN	ERRTST	RAD50
CLOSE	FDBSET	R50ASC
ERRSET	IRAD50	USEREX

  When porting existing programs, calls to ASSIGN, CLOSE, and FBDSET should be replaced with the appropriate OPEN statement. (You might consider converting DEFINE FILE statements at the same time, even though Digital Fortran for OpenVMS Alpha does support the DEFINE FILE statement.)

  In place of ERRSET and ERRTST, OpenVMS condition handling might be used. Note that Digital Fortran for OpenVMS Alpha supports the ERRSNS subroutine.

- Radix–50 constants in the form *n*R*xxx*

  For existing programs being ported, radix-50 constants and the IRAD50, RAD50, and R50ASC routines should be replaced by data encoded in ASCII using CHARACTER declared data.

Certain DEC Fortran for OpenVMS VAX Systems features have restricted use or are not available in Digital Fortran for OpenVMS Alpha:

- Numeric local variables are sometimes, but not always, initialized to a zero value, depending on the level of optimization used. To guarantee that a value will be initialized to zero under all circumstances, use an explicit assignment or DATA statement.
- Character constants must be associated with character dummy arguments, not numeric dummy arguments. (DEC Fortran for OpenVMS VAX Systems passed `'A'` by reference if the dummy argument was numeric.) Consider using the /BY_REF_CALL qualifier for such arguments.
- Saved dummy arrays do not work:

```
SUBROUTINE F_INIT (A, N)
REAL A(N)
RETURN
ENTRY F_DO_IT (X, I)
A (I) = X  ! No: A no longer visible
RETURN
END
```

- Hollerith actual arguments must be associated with numeric dummy (formal) arguments, not character dummy arguments.

The following language features are available in DEC Fortran for OpenVMS VAX Systems but are not supported in Digital Fortran for OpenVMS Alpha because of differences between the Alpha architecture and the VAX architecture:

- Certain FORSYSDEF symbol definition modules may be specific to the VAX or Alpha architecture.
- Precise exception-handling control

  The handling of certain exceptions differs between OpenVMS VAX and OpenVMS Alpha systems. To request precise exception-handling control, use the /SYNCHRONOUS_EXCEPTIONS qualifier.
- REAL*16 data uses the H_floating data format on VAX systems and X_floating on Alpha systems.
- VAX support for D_floating

  Because the Alpha instruction set does not support the D_floating REAL*8 format, D_floating data is converted to G_floating by software during computations and then converted back to D_floating format. Thus, there will be differences in D_floating arithmetic between VAX and Alpha systems.

  For optimal performance on Alpha systems, consider using REAL*8 data in VAX G_floating or IEEE T_floating format, perhaps using the /FLOAT qualifier to specify the format. To create a Digital Fortran for OpenVMS Alpha application program to convert D_floating data to G_floating or T_floating format, use the file conversion methods described in the Fortran language reference manual.
- Vectorization capabilities

  Vectorization, including /VECTOR and its related qualifiers, and the CDEC$ INIT_DEP_FWD directive are not supported. The Alpha processor provides pipelining and other features that resemble vectorization capabilities.

12.4.1.3 Interpretation Differences

The following language features are interpreted differently between DEC Fortran for OpenVMS VAX Systems and Digital Fortran for OpenVMS Alpha:

- Random number generator (RAN)

  The RAN function generates a different pattern of numbers in Digital Fortran for OpenVMS Alpha than in DEC Fortran for OpenVMS VAX Systems for the same random seed. (The RAN and RANDU functions are provided for DEC Fortran for OpenVMS VAX Systems compatibility.)
- Hollerith constants in formatted I/O statements

  DEC Fortran for OpenVMS VAX Systems and Digital Fortran for OpenVMS Alpha behave differently if either of the following occurs:

  - Two different I/O statements refer to the same CHARACTER PARAMETER constant as their format specifier. For example:

    ```
    CHARACTER*(*) FMT2
    PARAMETER (FMT2='(10Habcdefghij)')
    READ (5, FMT2)
    WRITE (6, FMT2)
    ```

- Two different I/O statements use the identical character constant as their format specifier. For example:

```
READ (5, '(10Habcdefghij)')
WRITE (6, '(10Habcdefghij)')
```

In DEC Fortran for OpenVMS VAX Systems, the value obtained by the READ statement is the output of the WRITE statement (FMT2 is ignored). However, in Digital Fortran for OpenVMS Alpha, the output of the WRITE statement is "abcdefghij." (The value read by the READ statement has no effect on the value written by the WRITE statement.)

12.4.2 Command Line Qualifiers

While Digital Fortran for OpenVMS Alpha and DEC Fortran for OpenVMS VAX Systems share most qualifiers, some qualifiers are specific to each platform. This section summarizes the differences between Digital Fortran for OpenVMS Alpha and DEC Fortran for OpenVMS VAX Systems command line qualifiers.

For complete details about the Digital Fortran for OpenVMS Alpha compilation command and options, see the *DEC Fortran User Manual for OpenVMS AXP Systems*. For complete details about the DEC Fortran for OpenVMS VAX Systems compilation command and options, see the *DEC Fortran User Manual for OpenVMS VAX Systems*.

To initiate compilation on either VAX or Alpha systems, use the FORTRAN command.

12.4.2.1 Qualifiers Specific to Digital Fortran for OpenVMS Alpha

Table 12–10 lists Digital Fortran for OpenVMS Alpha compiler qualifiers that have no equivalent DEC Fortran for OpenVMS VAX Systems options and are not supported in DEC Fortran for OpenVMS VAX Systems Version 6.3.

Table 12–10 Digital Fortran for OpenVMS Alpha Qualifiers Not in DEC Fortran for OpenVMS VAX Systems

Qualifier	Description
/BY_REF_CALL	Allows character constant actual arguments to be associated with numeric dummy arguments (allowed by DEC Fortran for OpenVMS VAX Systems).
/CHECK=FP_EXCEPTIONS	Controls whether messages about IEEE floating-point exceptional values are reported at run time.
/DOUBLE_SIZE	Makes DOUBLE PRECISION declarations REAL*16 instead of REAL*8.
/FAST	Sets several qualifiers that improve run-time performance.
/FLOAT	Controls the format used for floating-point data (REAL or COMPLEX) in memory, including the selection of either VAX F_floating or IEEE S_floating for KIND=4 data and VAX G_floating, VAX D_floating, or IEEE T_floating for KIND=8 data. DEC Fortran for OpenVMS VAX Systems provides the /[NO]G_FLOATING qualifier.

(continued on next page)

Table 12–10 (Cont.) Digital Fortran for OpenVMS Alpha Qualifiers Not in DEC Fortran for OpenVMS VAX Systems

Qualifier	Description
/GRANULARITY	Controls the granularity of data access for shared data.
/IEEE_MODE	Controls how floating-point exceptions are handled for IEEE data.
/INTEGER_SIZE	Controls the size of INTEGER and LOGICAL declarations.
/NAMES	Controls whether external names are converted to uppercase, lowercase, or as is.
/OPTIMIZE	The /OPTIMIZE qualifier supports the INLINE keyword, the TUNE keyword, the UNROLL keyword, and software pipelining.
/REAL_SIZE	Controls the size of REAL and COMPLEX declarations.
/ROUNDING_MODE	Controls how floating-point calculations are rounded for IEEE data.
/SEPARATE_COMPILATION	Controls whether the DEC Fortran compiler: • Places individual compilation units as separate modules in the object file like DEC Fortran for OpenVMS VAX Systems (/SEPARATE_COMPILATION) • Groups compilation units as a single module in the object file (/NOSEPARATE_COMPILATION, the default), which allows more interprocedure optimizations.
/SYNTAX_ONLY	Requests that only syntax checking occurs and no object file is created.
/WARNINGS	Certain keywords are not available on DEC Fortran for OpenVMS VAX Systems.
/VMS	Requests that Digital Fortran use certain DEC Fortran for OpenVMS VAX Systems conventions.

12.4.2.2 Qualifiers Specific to DEC Fortran for OpenVMS VAX Systems

This section summarizes DEC Fortran for OpenVMS VAX Systems compiler qualifiers that have no equivalent Digital Fortran for OpenVMS Alpha qualifiers.

Table 12–11 lists compilation qualifiers specific to DEC Fortran for OpenVMS VAX Systems Version 6.3.

Table 12–11 DEC Fortran for OpenVMS VAX Systems Qualifiers Not in Digital Fortran for OpenVMS Alpha

DEC Fortran for OpenVMS VAX Systems Qualifier	Description
/BLAS=(INLINE,MAPPED)	Specifies whether DEC Fortran for OpenVMS VAX Systems recognizes and inlines or maps the Basic Linear Algebra Subroutines (BLAS). Available only in DEC Fortran for OpenVMS VAX Systems.
/CHECK=ASSERTIONS	Enables or disables assertion checking. Available only in DEC Fortran for OpenVMS VAX Systems.
/DESIGN=[NO]COMMENTS /DESIGN=[NO]PLACEHOLDERS	Analyzes program for design information.
/DIRECTIVES=DEPENDENCE	Specifies whether specified compiler directives are used at compilation. Available only in DEC Fortran for OpenVMS VAX Systems.
/PARALLEL=(MANUAL or AUTOMATIC)	Supports parallel processing.
/SHOW=(DATA_DEPENDENCIES,DICTIONARY,LOOPS)	Control whether the listing file includes: • Diagnostics about loops that are ineligible for dependence analysis and data dependencies that inhibit vectorization or autodecomposition (DATA_DEPENDENCIES) • Source lines from included Common Data Dictionary records (DICTIONARY) • Reports about loop structures after compilation (LOOPS) The keywords DATA_DEPENDENCIES and LOOPS are available only in DEC Fortran for OpenVMS VAX Systems.
/VECTOR	Requests vector processing. Available only in DEC Fortran for OpenVMS VAX Systems.
/WARNINGS=INLINE	Controls whether the compiler prints informational diagnostic messages when it is unable to generate inline code for a reference to an intrinsic routine. Available only in DEC Fortran for OpenVMS VAX Systems.

All CPAR$ directives and certain CDEC$ directives associated with directed (manual) decomposition and their associated qualifiers or keywords are specific to DEC Fortran for OpenVMS VAX Systems, as described in the *DEC Fortran Language Reference Manual.*

For details about the DEC Fortran for OpenVMS VAX Systems compilation commands and options, see the *DEC Fortran User Manual for OpenVMS VAX Systems.*

12.4.3 Interoperability with Translated Shared Images

Using Digital Fortran for OpenVMS Alpha, you can create images that can interoperate with translated images at image activation (run time).

To allow the use of translated shared images:

- On the FORTRAN command line, specify the /TIE qualifier.
- On the LINK command line, specify the /NONATIVE_ONLY qualifier.

The created executable image contains code that allows the resulting executable image to interoperate with shared images, including allowing the DEC Fortran for OpenVMS VAX Systems RTL (FORRTL) to work with the Digital Fortran for OpenVMS Alpha RTL (DEC$FORTRTL). The native (Digital Fortran for OpenVMS Alpha RTL) and translated (DEC Fortran for OpenVMS VAX Systems RTL) programs can perform I/O to the same unit number, as long as the RTL that opens the file also closes it.

Programs should use the intrinsic names (without the prefix) rather than calling routines by their complete (*fac$xxxx*) name. One allowable exception to using *fac$xxxx* names is that translated image programs declare the FOR$RAB system function as EXTERNAL. Native Alpha programs should use FOR$RAB as an intrinsic function.

12.4.4 Porting DEC Fortran for OpenVMS VAX Systems Data

Record types are identical for DEC Fortran for OpenVMS VAX Systems and Digital Fortran for OpenVMS Alpha. If needed, transport the data using the EXCHANGE command with the /NETWORK and /TRANSFER=BLOCK qualifiers. To convert the file to Stream_LF format during the copy operation, use /TRANSFER=(BLOCK,RECORD_SEPARATOR=LF) instead of /TRANSFER=BLOCK, or specify the /FDL qualifier to the EXCHANGE command to change the record type or other file characteristics.

If you need to convert unformatted floating-point data, keep in mind that DEC Fortran for OpenVMS VAX programs (VAX hardware) store REAL*4 or COMPLEX*8 data in F_floating format, REAL*8, REAL*16, or COMPLEX*16 data in either D_floating or G_floating format, and REAL*16 data in H_floating format. Digital Fortran for OpenVMS Alpha programs (running on Alpha hardware) store REAL*4, REAL*8, REAL*16, COMPLEX*8, and COMPLEX*16 data in one of the formats shown in Table 12–12.

Table 12–12 Floating-Point Data on VAX and Alpha Systems

Data Declaration	VAX Formats	Alpha Formats
REAL*4 and COMPLEX*8	VAX F_floating format	IEEE S_floating or VAX F_floating format
REAL*8 and COMPLEX*16	VAX D_floating or G_ floating format	IEEE T_floating, VAX D_floating[1], or VAX G_floating format
REAL*16	VAX H_floating	X_floating. Requires conversion, perhaps using the /CONVERT qualifier or associated OPTION statement, logical name, or OPEN statement /CONVERT keyword. You can also use the RTL routine CVT$CONVERT_FLOAT.

[1]On Alpha systems, the use of VAX D_floating format involving many computations is not recommended. Consider converting D_floating format to IEEE T_floating (or VAX G_floating) format in a conversion program that uses the Digital Fortran for OpenVMS Alpha conversion routines.

12.5 Compatibility of DEC Pascal for OpenVMS Alpha Systems with VAX Pascal

This section compares DEC Pascal to other Digital Pascal compilers and lists the differences between DEC Pascal on VAX and Alpha systems. For a complete description of these features, see the *DEC Pascal Language Reference Manual*.

12.5.1 New Features of DEC Pascal

Table 12–13 lists features not previously supplied in VAX Pascal.

Table 12–13 New Features of DEC Pascal

Feature	Description
Support for OpenVMS systems	Including all the data types available on the OpenVMS platforms.
Redefinable values for predeclared constants	Values for MAXINT, MAXUNSIGNED, MAXREAL, MINREAL, EPSREAL are defined by the platform and the compiler switches for specifying the integer size and floating-point format.
An optional quoted parameter to the COMMON, EXTERNAL, GLOBAL, PSECT, WEAK_EXTERNAL, and WEAK_GLOBAL attributes	Allows you to pass an unmodified identifier to the linker.
Double-quoted strings	DEC Pascal now accepts the double-quote characters as string and character delimiters.
Embedded string values	Inside of double-quoted strings, DEC Pascal now supports constant characters specified with a backslash as in the C programming language, such as “"\n"” for the linefeed character.
Additional data types and values	DEC Pascal now supports these data types: ALFA, CARDINAL, CARDINAL16, CARDINAL32, INTEGER16, INTEGER32, INTEGER64, INTSET, POINTER, UNIV_PTR, UNSIGNED16, UNSIGNED32, and UNSIGNED64.
Assignment of UNSIGNED values to INTEGER variables	DEC Pascal now allows UNSIGNED values to be assignment-compatible with INTEGER variables and array indices.
Assignment of string values into unpacked arrays of characters	DEC Pascal now allows ARRAY of CHAR variables to be treated as fixed-length character strings.
Additional statements	DEC Pascal now supports these statements: BREAK, CONTINUE, EXIT, NEXT, and RETURN.
Additional predeclared routines	DEC Pascal now supports these functions and procedures: ADDR, ARGC, ARGV, ASSERT, BITAND, BITNOT, BITOR, BITXOR, HBOUND, LBOUND, FIRST, FIRSTOF, LAST, LASTOF, IN_RANGE, LSHIFT, RSHIFT, LSHFT, RSHFT, MESSAGE, NULL, RANDOM, SEED, REMOVE, SIZEOF, SYSCLOCK, and WALLCLOCK.
Optional second parameter to RESET, REWRITE, and EXTEND	DEC Pascal now accepts a second parameter that is a literal string expression for the file name to be associated with the file variable.

(continued on next page)

Table 12–13 (Cont.) New Features of DEC Pascal

Feature	Description
Compiler command switches	DEC Pascal now includes switches that allow you to specify the storage and alignment allocation for data types. You can also specify the level of optimization with a switch. On Alpha systems, an option controls the default meaning of the REAL and DOUBLE data types. Arguments to the usage switch enable messages relating to alignment, alignment compatibility on different platforms, and features that are not available on a specified platform.

12.5.2 Establishing Dynamic Condition Handlers

DEC Pascal provides the built-in routines, ESTABLISH and REVERT, to use in place of LIB$ESTABLISH. If you declare and try to use LIB$ESTABLISH, you will get a compile-time warning.

12.5.3 Modifying Default Alignment Rules for Record Fields

DEC Pascal allows you to override field alignment and position with the POS, ALIGNED, and DATA attributes and the data compiler switch.

12.5.4 Recommended Use of Predeclared Identifiers

Although for backward compatibility DEC Pascal compiles programs that include the predeclared identifiers listed in Table 12–14, Digital recommends that you use the listed replacements.

Table 12–14 Recommended Use of Predeclared Identifiers

Identifier	Recommended Usage
ADDR	Use the ADDRESS function
ALFA	Equivalent to TYPE ALFA = PACKED ARRAY [1..10]OF CHAR
BITAND	Equivalent to the UAND statement
BITNOT	Equivalent to the UNOT statement
BITOR	Equivalent to the UOR statement
BITXOR	Equivalent to the UXOR statement
EXIT	Equivalent to the BREAK statement
FIRST, FIRSTOF	Equivalent to the LOWER function
HBOUND	Equivalent to the UPPER function
IN_RANGE	Useful only when subrange checking is disabled. IN_RANGE(X) is equivalent to (X≥LOWER(X))AND(X≤UPPER(X)).
INTSET	Equivalent to TYPE INTSET = SET OF 0 .. 255;
LAST, LASTOF	Equivalent to the UPPER function
LBOUND	Equivalent to the LOWER function
LSHFT	Equivalent to the LSHIFT function
MESSAGE	Equivalent to WRITELN(ERR,expression)

(continued on next page)

Table 12–14 (Cont.) Recommended Use of Predeclared Identifiers

Identifier	Recommended Usage
NEXT	Equivalent to the CONTINUE statement
NULL	Equivalent to the empty statement
REMOVE	Equivalent to the DELETE_FILE procedure
RSHFT	Equivalent to the RSHIFT function
SIZEOF	Equivalent to the SIZE function
STLIMIT	Compiles but does not return an error
UNIV_PTR	Equivalent to TYPE UNIV_PTR = POINTER;

12.5.5 Platform-Dependent Features

DEC Pascal can use an environment file only on the same platform (the combination of operating system and hardware) on which it was compiled.

In addition, the following lists features of DEC Pascal supplied only on VAX systems:

- QUADRUPLE data type
- H_floating-point data type
- VAX Pascal Version 1.0 dynamic arrays
- MFPR and MTPR predeclared routines
- [OVERLAID] attribute
- Table of contents in listing
- Optimize attribute on routines

The following lists features of DEC Pascal supplied only on Alpha systems:

- Abbreviations when reading enumerated data types
- Indexed file organization
- Relative file organization

12.5.6 Obsolete Features

This section describes features that are supported, but not recommended, by Digital. They are provided only for compatibility with other Digital Pascal compilers.

12.5.6.1 /OLD_VERSION Qualifier

The /OLD_VERSION qualifier directed the compiler to resolve differences between VAX Pascal Version 1.0 and subsequent versions by using the VAX Pascal Version 1.0 definition of the language. The qualifier is provided so that existing programs continue to work.

12.5.6.2 /G_FLOATING Qualifier

The /G_FLOATING qualifier directs the compiler to use the G_floating representation and instructions for values of type DOUBLE. The [[NO]G_FLOATING] attribute can be specified on both OpenVMS VAX and OpenVMS Alpha systems.

If the use of the /G_FLOATING qualifier conflicts with a double-precision attribute specified in the source program or module, an error occurs. Routines and compilation units between which double-precision quantities are passed should not mix floating-point formats. Not all OpenVMS VAX processors support the G_floating data types.

See also the description of the /FLOAT qualifier, which is the preferred method for specifying the floating-point format to the compiler. The /FLOAT qualifier also allows you to select the IEEE floating-point format, which is supported only on Alpha systems.

12.5.6.3 OVERLAID Attribute

The OVERLAID attribute indicates how storage should be allocated for variables declared within a compilation unit. If you specify OVERLAID on a compilation unit, the variables declared at program or module level (unless they have the STATIC or PSECT attribute) overlay the storage of static variables in all other overlaid compilation units.

This attribute is intended for use only with programs that use the decommitted separate compilation facility provided by VAX Pascal Version 1.0.

A

Application Evaluation Checklist

This checklist is based on one used by Digital to evaluate applications for OpenVMS Alpha.

Comments in brackets following a question are intended to help clarify the purpose of that question.

Application Evaluation Checklist

Development History and Plans

1. Does the application currently run on other operating systems or hardware architectures? ☐ YES ☐ NO

   If yes, does the application currently run on a RISC system? ☐ YES ☐ NO

   [If so, it will be easier to migrate to OpenVMS Alpha.]

2. What are your plans for the application after migration?

   a. No further development ☐ YES ☐ NO

   b. Maintenance releases only ☐ YES ☐ NO

   c. Additional or changed functionality ☐ YES ☐ NO

   d. Maintain separate VAX and Alpha sources ☐ YES ☐ NO

   [If you answer YES to a, you may wish to consider translating the application. A YES response to b or c should give you reason to evaluate the benefits of recompiling and relinking your application, although translation is still possible. If you intend to maintain separate VAX and Alpha sources, as indicated by a YES to d, you may need to consider interoperability and consistency issues, especially if the different versions of the application can access the same database.]

External Dependencies

3. What is the system configuration (CPUs, memory, disks) required to set up a development environment for the application? ________________

   [This will help you plan for the resources needed for migration.]

4. What is the system configuration (CPUs, memory, disks) required to set up a typical user environment for the application, including installation verification procedures, regression tests, benchmarks, or workloads? ________________

   [This will help you determine whether your entire environment is available on OpenVMS Alpha.]

5. Does the application rely on any special hardware? ☐ YES ☐ NO

   [This will help you determine whether the hardware is available on OpenVMS Alpha, and whether the application includes hardware-specific code.]

6. a. What version of OpenVMS does your application currently run on? ________________

   b. Does the application run on OpenVMS VAX Version 7.0? ☐ YES ☐ NO

c. Does the application use features that are not available on OpenVMS Alpha? ☐ YES ☐ NO

[The migration base for OpenVMS Alpha is OpenVMS VAX Version 7.0. If you answer YES to c, your application may use features that are not yet supported on OpenVMS Alpha, or be linked against an OpenVMS RTL or other shareable image that is incompatible with the current version of OpenVMS Alpha.]

7. Does the application require layered products to run?

a. From Digital: (other than compiler RTLs) ☐ YES ☐ NO

b. From third parties: ☐ YES ☐ NO

[If you answer YES to a and are uncertain whether the Digital layered products are yet available for OpenVMS Alpha, check with your Digital Account Representative. If you answer YES to b, check with your third-party product supplier.]

Composition of the Application

8. How large is your application?

How many modules? ____________

How many lines or kilobytes of code? ____________

How much disk space is required? ____________

[This will help you "size" the effort and the resources required for migration.]

9. a. Do you have access to all source files that make up your application? ☐ YES ☐ NO

b. If you are considering using Digital Services, will it be possible to give Digital access to these source files and build procedures? ☐ YES ☐ NO

[If you answer YES to a, translation may be your only migration option for the files with missing sources. A YES answer to b allows you to take advantage of a greater range of Digital migration services.]

10. a. What languages is the application written in? (If multiple languages are used, give the percentages of each.) ____________

[If the compilers are not yet available, you must translate or rewrite in a different language.]

b. If you use VAX MACRO, what are your specific reasons? ____________

c. Could the function of the VAX MACRO code be performed by a high-level-language compiler or a system service (such as $GETJPI for retrieving process names)? ☐ YES ☐ NO

[Digital does not recommend the use of VAX MACRO or the MACRO–64 Assembler for OpenVMS Alpha in Alpha applications. You may be able to replace assembly-language code in certain user-mode applications by a call to an OpenVMS system service that did not exist when the application was first written.]

11. a. Do you have regression tests for the application? ☐ YES ☐ NO

    b. If yes, do they require DEC Test Manager? ☐ YES ☐ NO

    [If you answer YES to a, you should consider migrating those regression tests. The DEC Test Manager is not available at the initial release of OpenVMS Alpha. Contact your Digital Account Representative if your regression tests depend on this product.]

Dependencies on the VAX Architecture

12. a. Does the application use the H_floating data types? ☐ YES ☐ NO

    b. Does the application use the D_floating data types? ☐ YES ☐ NO

    c. If the application uses D_floating, does it require 56 bits of precision (16 decimal digits) or would 53 bits (15 decimal digits) suffice? ☐ 56 bits ☐ 53 bits

    [If you answer YES to a, you must either translate your application to obtain H_floating compatibility, or convert the data to G_floating, S_floating, or T_floating format. If you answer YES to b, you must either translate the application to obtain full 56-bit VAX precision D_floating compatibility, accept the 53-bit precision D_floating format provided by Alpha systems, or convert the data to G_floating, S_floating, or T_floating format.]

13. a. Does the application use large amounts of data or data structures? ☐ YES ☐ NO

    b. Is the data byte, word, or longword aligned? ☐ YES ☐ NO

    [If you answer YES to a, but NO to b, you should consider aligning your data naturally to achieve optimal Alpha performance. You must align data naturally if the data is in a global section shared among a number of processes, or is shared between a main program and an AST.]

14. Does the application make assumptions about how compilers align data (that is, does the application assume that data structures are: packed, aligned naturally, aligned on longwords, and so forth)? ☐ YES ☐ NO

    [If you answer YES, you should consider portability and interoperability issues resulting from differences in compiler behavior, both on the Alpha platform and between the VAX and Alpha platforms. Be aware that compiler defaults for data alignment vary, as do compiler switches for forcing alignment. Typically, VAX systems default to a packed style alignment, whereas Alpha compilers default to natural alignment where possible.]

15. a. Does the application assume a 512-byte page size? ☐ YES ☐ NO

b. Does the application assume that a memory page is the same size as a disk block? ☐ YES ☐ NO

[If you answer YES to a, you should be prepared to adapt the application to accommodate the Alpha page size, which is much larger than 512 bytes and varies from system to system. Avoid hardcoded references to the page size; rather, use memory management system services and RTL routines wherever possible. If you answer YES to b, you should examine all calls to the $CRMPSC and $MGBLSC system services that map disk sections to memory and remove these assumptions.]

16. Does the application call OpenVMS system services? ☐ YES ☐ NO

Specifically, services that:

a. Create or map global sections (such as $CRMPSC, $MGBLSC, $UPDSEC) ☐ YES ☐ NO

b. Modify the working set (such as $LCKPAG, $LKWSET) ☐ YES ☐ NO

c. Manipulate virtual addresses (such as $CRETVA, $DELTVA) ☐ YES ☐ NO

[If you answer YES to any of these, you may need to examine your code to determine that it specifies the required input parameters correctly.]

17. a. Does the application use multiple, cooperating processes? ☐ YES ☐ NO

If so:

b. How many processes? ________________

c. What interprocess communication method is used? ________________

☐ $CRMPSC	☐ Mailboxes	☐ SCS	☐ Other
☐ DLM	☐ SHM, IPC	☐ SMG$	☐ STR$

d. If you use global sections ($CRMPSC) to share data with other processes, how is data access synchronized? ________________

[This will help you determine whether you will need to use explicit synchronization, and the level of effort required to guarantee synchronization among the parts of your application. Use of a high-level synchronization method generally allows you to migrate an application most easily.]

18. Does the application currently run in a multiprocessor (SMP) environment? ☐ YES ☐ NO

[If you answer YES, it is likely that your application already uses adequate interprocess synchronization methods.]

19. Does the application use AST (asynchronous system trap) mechanisms? ☐ YES ☐ NO

[If you answer YES, you should determine whether the AST and main process share access to data in process space. If so, you may need to explicitly synchronize such accesses.]

20. a. Does the application contain condition handlers? ☐ YES ☐ NO

b. Does the application rely on immediate reporting of arithmetic exceptions? ☐ YES ☐ NO

[The Alpha architecture does not provide immediate reporting of arithmetic exceptions. If your handler attempts to fix the condition and restart the instruction sequence that led to the exception, you will need to alter the handler.]

21. Does the application run in privileged mode or link against SYS.STB? ☐ YES ☐ NO

If so, why? ________________

[If your application links against the OpenVMS executive or runs in privileged mode, you must rewrite it for it to work as a native Alpha image.]

22. Do you write your own device drivers? ☐ YES ☐ NO

[User-written device drivers are not supported in the initial release of OpenVMS Alpha. Contact your Digital Account Representative if you need this feature.]

23. Does the application use connect-to-interrupt mechanisms? ☐ YES ☐ NO

If yes, with what functionality? ________________

[Connect-to-interrupt is not supported on OpenVMS Alpha systems. Contact your Digital Account Representative if you need this feature.]

24. Does the application create or modify machine instructions? ☐ YES ☐ NO

[Guaranteeing correct execution of instructions written to the instruction stream requires great care on OpenVMS Alpha.]

25. What parts of the application are most sensitive to performance? I/O, floating point, memory, realtime (that is, interrupt latency, and so on). ________________

[This will help you determine how to prioritize work on the various parts of your application and allow Digital to plan performance enhancements that are most meaningful to customers.]

Glossary

alignment

See *natural alignment*.

atomic instruction

An instruction that consists of one or more discrete operations that are handled by the hardware as a single operation, without interruption.

atomic operation

An operation that cannot be interrupted by other system events, such as an AST (asynchronous system trap) service routine; an atomic operation appears to other processes to be a single operation. Once an atomic operation starts, it always completes without interruption.

Read-modify-write operations are typically not atomic at an instruction level on a RISC machine.

byte granularity

A property of memory systems in which adjacent bytes can be written concurrently and independently by different processes or processors.

CISC

See *complex instruction set computer.*

compatibility

The ability of programs written for one type of computer system (such as OpenVMS VAX) to execute on another type of system (such as OpenVMS Alpha).

complex instruction set computer (CISC)

A computer that has individual instructions that perform complex operations, including complex operations performed directly on locations in memory. Examples of such operations include instructions that do multibyte data moves or substring searches. CISC computers are typically contrasted with *RISC* (*reduced instruction set computer*) computers.

concurrency

Simultaneous operations by multiple agents on a shared object.

cross development

The process of creating software using tools running on one system, but targeted for another type of system; for example, creating code for Alpha systems using tools running on a VAX system.

granularity

A characteristic of storage systems that defines the amount of data that can be read or written with a single instruction, or read or written independently. VAX systems have byte or multibyte granularities while disk systems typically have 512-byte or greater granularities.

image information file (IIF)

An ASCII file that contains information about the interface between VAX images. VEST uses IIFs to resolve references to other images and to generate the appropriate linkages.

image section

A group of program sections with the same attributes (such as read-only access, read/write access, absolute, relocatable, and so on) that is the unit of virtual memory allocation for an image.

interlocked instruction

An instruction that performs some action in a way that guarantees the complete result as a single, uninterruptible operation in a multiprocessing environment. Since other potentially conflicting operations can be blocked while the interlocked instruction completes, interlocked instructions can have a negative performance impact.

jacket routine

A procedure that converts procedure calls from one calling standard to another; for example, calls between translated VAX images, which use the VAX calling standard, and native Alpha images, which use the Alpha calling standard.

load/store architecture

A machine architecture in which data items are first loaded into a processor register, then operated on, and finally stored back to memory. No operations on memory other than load and store are provided by the instruction set.

longword

Four contiguous bytes (32 bits) starting on any addressable byte boundary. Bits are numbered from right to left, 0 to 31. The address of the longword is the address of the byte containing the low-order bit (bit 0). A longword is *naturally aligned* if its address is evenly divisible by 4.

multiple instruction issue

Issuing more than one instruction during a single clock cycle.

natural alignment

Data storage in memory such that the address of the data is evenly divisible by the size of the data in bytes. For example, a naturally aligned longword has an address that is evenly divisible by 4, and a naturally aligned quadword has an address that is evenly divisible by 8. A structure is naturally aligned when all its members are naturally aligned.

page size

The number of bytes that a system's hardware treats as a unit for address mapping, sharing, protection, and movement to and from secondary storage.

pagelet

A 512-byte unit of memory in an Alpha environment. On Alpha systems, certain DCL and utility commands, system services, and system routines accept as input or provide as output memory requirements and quotas in terms of pagelets. Although this allows the external interfaces of these components to be compatible with those of VAX systems, OpenVMS Alpha internally manages memory only in even multiples of the CPU memory page size.

PALcode

See *privileged architecture library*.

privileged architecture library (PAL)

A library of callable routines for performing instructions unique to a particular operating system. Special instructions call the routines, which must run without interruption.

processor status (PS)

On Alpha systems, a privileged processor register consisting of a *quadword* of information including the current access mode, the current interrupt priority level (IPL), the stack alignment, and several reserved fields.

processor status longword (PSL)

On VAX systems, a privileged processor register consisting of a word of privileged processor status and the *processor status word* itself. The privileged processor status information includes the current interrupt priority level (IPL), the previous access mode, the current access mode, the interrupt stack bit, the trace trap pending bit, and the compatibility mode bit.

processor status word (PSW)

On VAX systems, the low-order word of the *processor status longword*. Processor status information includes the condition codes (carry, overflow, 0, negative), the arithmetic trap enable bits (integer overflow, decimal overflow, floating underflow), and the trace enable bit.

program counter (PC)

That portion of the CPU that contains the virtual address of the next instruction to be executed. Most current CPUs implement the program counter as a register. This register is visible to the programmer through the instruction set.

quadword

Four contiguous words (64 bits) starting on any addressable byte boundary. Bits are numbered from right to left, 0 to 63. The address of a quadword is the address of the word containing the low-order bit (bit 0). A quadword is *naturally aligned* if its address is evenly divisible by 8.

quadword granularity

A property of memory systems in which adjacent *quadwords* can be written concurrently and independently by different processes or processors.

read-modify-write operation

A hardware operation that involves the reading, modifying, and writing of a piece of data in main memory as a single, uninterruptible operation.

read-write ordering

The order in which memory on one CPU becomes visible to an execution agent (a different CPU or device within a tightly coupled system).

reduced instruction set computer (RISC)

A computer that has an instruction set reduced in complexity, but not necessarily in the number of instructions. RISC architectures typically require more instructions than *CISC* architectures to perform a given operation, because an individual instruction performs less work than a CISC instruction.

RISC

See *reduced instruction set computer.*

synchronization

A method of controlling access to some shared resource so that predictable, well-defined results are obtained when operating in a multiprocessing environment or in a uniprocessing environment using shared data.

translated code

The native Alpha object code in a translated image. Translated code includes:

- Alpha code that reproduces the behavior of equivalent VAX code in the original image
- Calls to the *Translated Image Environment (TIE)*

translated image

An Alpha executable or shareable image created by *translation* of the object code of a VAX image. The translated image, which is functionally equivalent to the VAX image from which it was translated, includes both *translated code* and the original image. See *VAX Environment Software Translator*.

Translated Image Environment (TIE)

A native Alpha shareable image that supports the execution of *translated images*. The TIE processes all interactions with the native Alpha system and provides an environment similar to OpenVMS VAX for the translated image by managing VAX state; by emulating VAX features such as exception processing, AST delivery, and complex VAX instructions; and by interpreting untranslated VAX instructions.

translation

The process of converting a VAX binary image to an Alpha image that runs with the assistance of the TIE on an Alpha system. Translation is a static process that converts as much VAX code as possible to native Alpha instructions. The TIE interprets any untranslated VAX code at run time.

VEST

See *VAX Environment Software Translator*.

VAX Environment Software Translator (VEST)

A software migration tool that performs the *translation* of VAX executable and shareable images into translated images that run on Alpha systems. See *translated image*.

word granularity

A property of memory systems in which adjacent words can be written concurrently and independently by different processes or processors.

writable global section

A data structure (for example, FORTRAN global common) or shareable image section potentially available to all processes in the system for use in communicating between processes.

Index

A

Access modes
inner, 2–6
__ADD_ATOMIC_LONG built-in, 12–7
__ADD_ATOMIC_QUAD built-in, 12–7
$ADJWSL system service
page-size dependencies, 6–2
Alignment
See Data alignment
Allocating memory
by expanding virtual address space
page-size dependencies, 6–6
freeing allocated memory
page-size dependencies, 6–9
page-size dependencies, 6–6
reallocating existing virtual addresses
page-size dependencies, 6–8
specifying address ranges, 6–8
specifying page counts, 6–6
using the $CRETVA system service, 6–9
using the $EXPREG system service, 6–7
Alpha architecture
compared to other RISC architectures, 1–5 to 1–7
compared to VAX, 1–4
general description, 1–4
Alpha instructions
accessing from DEC C, 12–6
Analyze/Image utility (ANALYZE/IMAGE), 4–6
Analyze/Object utility (ANALYZE/OBJECT), 4–6
Analyzing an application, 2–18 to 2–20
__AND_ATOMIC_LONG built-in, 12–7
__AND_ATOMIC_QUAD built-in, 12–7
AP
See Argument pointer (AP)
Application Migration Detailed Analysis Service, 1–10
Application Migration Service, 1–10
Applications
analyzing, 2–18 to 2–20
assessing portability, 1–7
establishing baseline values for, 4–12
languages used, A–3
size, A–3
VAX dependency checklist, 2–7
Architecture
dependencies, 2–7
ARCH_NAME keyword
determining host architecture, 5–5
ARCH_TYPE keyword
determining host architecture, 5–4
Argument lists
accessing from DEC C, 12–7
Argument pointer (AP), 2–16
Arithmetic exceptions, 2–15 to 2–16
condition handler for, 2–16
on Alpha systems, 9–8
precise
VEST qualifiers, 2–24
Assembly language
no performance advantage on Alpha, 2–6
replaced by system services, 2–6
AST parameter list
reliance on architectural details of, 2–18
ASTs (asynchronous system traps), 1–6, A–6
sharing data, 2–11
synchronizing with, 2–12
AST service routines
dependence on parameter list, 2–8, 2–18
Asynchronous system traps
See ASTs
Atomic instructions
effect on synchronization, 7–2
Atomicity
DEC C support, 12–6
definition, 2–11
language constructs to guarantee, 2–12
of byte and word write operations, 2–12, 2–22
of read-modify-write operations, 2–22, 3–5
preserving in translated images, 7–10
provided by PALcode, 1–6
VEST qualifiers
instruction, 2–23
memory, 2–23

B

Based images, 2–5
Baseline values for application
establishing, 4–12

BASIC
translated images, 10–10
BLAS$ functions invoked by translated images, 10–10
BLAS1RTL translated library, 10–10
/BPAGE linker qualifier
linking VAX images to be translated, 2–24, 11–4
Buffer sizes
in mixed-architecture VMScluster system, 2–8
Bugs
latent, 4–13
Build procedures, 2–2
changes required, 1–1
Byte granularity, 2–12, 2–22
effect on synchronization, 7–2
specifying, 2–13

C

C
header files for defining macros, 4–5
LIB$ESTABLISH, 9–1, 12–8
Call frames
interpreting contents of, 2–16
Calling standard
call frame stack, 3–5
reliance on, 2–16
Calls
nonstandard
writing jacket routines for, 2–25
CALL*x* VAX instruction, 2–25
Choosing a migration method, 2–3, 2–22
CLUE (Crash Log Utility Extractor)
See Crash Log Utility Extractor
$CMEXEC system service, 2–6
$CMKRNL system service, 2–6
CMS (Code Management System), 2–2, 4–2
COBOL
fast performance, 2–10
packed decimal data, 2–10
COBOL programs support, 10–12
Code Management System
See CMS
Code reviews, 2–19
Command procedures, 1–1
Compatibility
granularity specified by compiler, 2–13
mixing native and translated images, 1–9
OpenVMS VAX and OpenVMS Alpha, 1–1 to 1–3
using translation for, 1–8, 2–23
Compile commands
changes required, 4–4
Compile procedures, 4–2
Compilers
architectural differences, 4–5
availability on Alpha, 2–3, 4–4
availability on Alpha systems, 5–1
BLISS, 4–10
commands, 4–4
compatibility between compilers on VAX systems and on Alpha systems, 12–1 to 12–39
data alignment defaults, 2–22
differences, 12–1
messages generated by, 2–19
native Alpha, 2–3, 4–4
optimizing, 4–4
options
exception reporting, 2–16
PALcode built-ins, 1–6
qualifiers, 1–1
qualifiers for VAX dependencies, 4–4
specifying granularity, 2–13
use of LIB$ESTABLISH routine, 9–1
Conditional compilation directives
DEC C incompatibility with VAX C, 12–9
Condition code
matching, 9–6
Condition handlers, 2–8, A–6
arithmetic exceptions, 2–16
establishing dynamic, 2–17, 9–1, 12–8, 12–29, 12–37
Condition handling
alignment fault reporting, 9–10
arithmetic exceptions, 9–8
condition codes, 9–6
enabling overflow detection, 9–12
hardware exception conditions, 9–7
mechanism array format, 9–3
on Alpha systems, 9–1
run-time library support routines, 9–11
signal array format, 9–2
specifying condition handlers, 9–12
unwinding, 9–5
VAX hardware exceptions, 9–7
with translated images, 9–7
writing condition handlers, 9–2
Connect-to-interrupt mechanisms, A–6
CPU keyword
determining the host architecture, 5–5
Crashes
analyzing, 4–11
Crash Log Utility Extractor (CLUE), 4–12
$CREPRC system service
page-size dependencies, 6–2
$CRETVA system service, A–5
code example, 6–9
page-size dependencies, 6–2
reallocating memory on an Alpha system, 6–8

$CRMPSC system service, 2–6, 2–14, A–5
  mapping a single page section
    page-size dependencies, 6–12
  mapping into a defined address range
    code example, 6–14
    page-size dependencies, 6–13
  page-size dependencies, 6–2
  used to map into expanded virtual address
    space
    code example, 6–11
    page-size dependencies, 6–10

D

Data
  See also Data alignment
  ODS-1 format not supported in Alpha, 1–2
  ODS-2 format unchanged, 1–2
  porting between Digital Fortran for OpenVMS
    Alpha and DEC Fortran for OpenVMS VAX
    Systems, 12–35
  shared
    access, 2–7
    unintentional sharing, 7–8
Data alignment, 2–9 to 2–10, 2–12, 2–22, A–4
  compiler defaults, 2–22
  compiler options, 2–9, 2–10
  DEC Ada support, 12–2
  DEC COBOL support, 12–14
    default alignment, 12–15
  DEC C support, 12–7
  DEC Pascal support, 12–37
  exception reporting, 9–10
  finding unaligned data, 2–9
  global sections, 2–5
  incompatibility with translated software, 2–10
  performance, 2–9, 2–22
  run-time faults, 2–20
  static unaligned data, 2–19
  unaligned stack operations, 2–19
  VEST qualifiers, 2–23
Databases
  same function on Alpha, 1–3
Data packing, 2–7
Data types, 2–10 to 2–11
  Alpha implementations, 2–10
  decimal, 2–10
  differences between Digital Fortran for
    OpenVMS Alpha and DEC Fortran for
    OpenVMS VAX Systems, 12–35
  D_floating, 1–6, 2–11, 2–19, 2–22
    full precision, 1–3, A–4
    in mixed-architecture clusters, 3–3, 3–4
  G_floating, 1–3, 2–11, 2–22
  H_floating, 1–3, 1–6, 2–7, 2–10, 2–19, 2–22,
    A–4
  IEEE formats, 2–11
    little endian, 1–2
Data types (cont'd)
  packed decimal, 2–19, 2–22
  portability between VAX and Alpha systems,
    8–1
  supported by Alpha architecture, 8–1
  supported by VAX architecture, 8–1
Data-type sizes
  DEC C portability macros, 12–5
  effect on protection of shared data, 7–9
  supported by DEC C, 12–5
DCL (DIGITAL Command Language), 1–1
Debugger, 4–8 to 4–11
  Delta/XDelta, 4–10
  detecting unaligned data, 2–10
  native Alpha, 4–6
  OpenVMS, 4–9
  System Code Debugger, 4–10
Debugging, 4–6, 4–8 to 4–11
  on Alpha hardware only, 4–8
  restrictions on Alpha systems, 4–9
  translated images, 4–10
DEC Ada
  compatibility with VAX Ada, 12–1
  language pragma support on Alpha systems,
    12–2
  system package support on Alpha systems,
    12–3
DEC C
  accessing Alpha instructions, 12–6
  accessing VAX instructions, 12–6
  ANSI conformance, 12–4
  atomicity built-ins, 12–6
  64-bit capabilities, 12–5
  compatibility modes, 12–4
  controlling data alignment, 12–7
  data-type-size portability macros, 12–5
  establishing dynamic condition handler, 12–8
  features specific to Alpha systems, 12–6
  specifying floating-point formats, 12–5
  /STANDARD qualifier, 12–4
  supported data-types, 12–5
  support for pcc mode, 12–4
  VAX C mode, 12–4
    incompatibilities with VAX C, 12–9
DEC C for OpenVMS Alpha systems
  See DEC C
DEC COBOL
  ACCEPT statement differences, 12–25
  /ALIGNMENT qualifier, 12–14
  /CHECK qualifier, 12–15
  command line qualifiers not supported by
    VAX COBOL, 12–12
  command line qualifiers shared with
    VAX COBOL, 12–11
  compatibility modes, 12–17
  compatibility with VAX COBOL, 12–10
  compiler messages, 12–20
  controlling data alignment, 12–14

DEC COBOL (cont'd)
/CONVERT=LEADING_BLANKS qualifier, 12–15
converting VAX COBOL programs, 12–20
COPY statement differences, 12–21
defining storage for return values, 12–27
differences in program structure, 12–20
DISPLAY statment differences, 12–25
EXIT PROGRAM statement, 12–18
file status differences, 12–26
/FLOAT qualifier, 12–16
I/O file status codes, 12–18
LINAGE statement differences, 12–25
listing file differences, 12–24
MOVE statement differences, 12–24
moving unsigned data items, 12–24
no valid next record condition, 12–19
/OPTIMIZE qualifier, 12–16
register set differences, 12–26
relationship to DEC SMG (Screen Manager), 12–25
REPLACE statement differences, 12–23
/RESERVED_WORDS qualifier, 12–16
RMS special registers, 12–27
/STANDARD=OPENVMS_Alpha qualifier option, 12–20
/STANDARD qualifier, 12–17
support for ANSI 1974 standard, 12–17
support for ANSI 1985 standard, 12–17
support for Version 3, 12–17
system return codes, 12–26
/TIE qualifier, 12–20
unreachable code analysis, 12–21
using data alignment directives, 12–15
validating numeric data, 12–15
/WARNINGS=STANDARD qualifier support, 12–19
WRITE statement, 12–25
X/Open reserved words list, 12–16
DECforms, 1–1
DEC Fortran for OpenVMS Alpha
compatibility with DEC Fortran for OpenVMS VAX Systems
architectural differences, 12–31
command line, 12–32
interpretation differences, 12–31
porting data, 12–35
restrictions, 12–30
establishing dynamic condition handler, 12–29
instrinsic names
prefixes, 12–35
interoperability considerations, 12–34
LIB$ESTABLISH routine, 9–1, 12–29
LIB$REVERT routine, 9–1, 12–29
performing I/O from native and translated images, 12–35
porting data, 12–35
DEC Fortran for OpenVMS Alpha (cont'd)
qualifiers not available in DEC Fortran for OpenVMS VAX Systems, 12–32
qualifiers specific to DEC Fortran for OpenVMS VAX Systems, 12–33
support for floating-point data types, 12–35
DECmigrate
See also Translated Image Environment (TIE) and Translated image support
support for translated images, 10–2
VEST, 10–2
DECmigrate utility
VEST command /PRESERVE qualifier, 7–10
DEC Pascal
compatibility with VAX Pascal, 12–38
differences with VAX Pascal, 12–35
establishing dynamic condition handler, 12–37
/G_FLOATING qualifier, 12–39
identifiers included for compatibility, 12–37
LIB$ESTABLISH routine, 9–1, 12–37
new features, 12–36
obsolete features, 12–38
/OLD_VERSION qualifier, 12–38
OVERLAID attribute, 12–39
specifying floating-point format, 12–39
support for data alignment, 12–37
DECset, 4–6
DECthreads
.H file support, 12–9
DECwindows, 1–1
Delta/XDelta Debugger (DELTA/XDELTA), 4–8
See also Debugger
OpenVMS Alpha, 4–10
$DELTVA system service, A–5
freeing allocated memory
page-size dependencies, 6–9
page-size dependencies, 6–3
Dependencies on other software
identifying, 2–1
$DEQ system service, 2–12, 2–15
Device configuration functions
in SYSMAN for Alpha, 1–2
Device drivers, 2–5
debugging, 4–10
Step 1 interface, 1–6
Step 2 interface, 1–6
user-written, 1–6, 2–6, A–6
written in C, 1–6
Diagnostic features
compilers, 2–18
VEST, 2–18
DIGITAL Command Language
See DCL
Digital Fortran for OpenVMS Alpha
compatibility with DEC Fortran for OpenVMS VAX Systems, 12–28
language features, 12–28

Digital Fortran for OpenVMS Alpha (cont'd)
differences with DEC Fortran for OpenVMS VAX Systems, 12–28
Digital Portable Mathematics Library
See DPML
Disk block size
relation to page size, 2–14
DMA controller, 2–13
DPML (Digital Portable Mathematics Library)
compatibility, 5–4
Dump files
See System dump files
Dynamic condition handler
establishing, 2–17
D_floating data type, 1–3, 1–6, 2–11, 2–19, 3–3
in mixed-architecture clusters, 3–4

E

Editors
unchanged for Alpha, 1–1
$ENQ system service, 2–12, 2–15
Evaluating code, 1–7
checklist, A–1
Exception handling
See Condition handling
Exception reporting, 2–15 to 2–16, 3–5, 3–7
compiler options, 2–16
immediacy of, A–6
imprecise, 2–15
precise, 2–15, 2–23
reliance on architectural details of, 2–17
Executive images
slicing, 4–11
$EXPREG system service
allocating memory on Alpha systems, 6–6
code example, 6–7
page-size dependencies, 6–3

F

File types
on Alpha systems, 5–2
Flag-passing protocols
for synchronization, 2–15
Floating-point data types
comparison of VAX and Alpha types, 2–7, 12–35
converting H_floating data, 12–35
CVT$CONVERT_FLOAT RTL routine, 12–35
DEC COBOL storage differences, 12–27
differences between DEC Fortran for OpenVMS VAX Systems and Digital Fortran for OpenVMS Alpha, 12–35
locating references, 2–19
specifying in DEC COBOL, 12–16
supported by DEC Ada, 12–2
Floating-point data types (cont'd)
supported by DEC C, 12–5
supported by DEC Pascal, 12–39
VAX little-endian formats, 12–35
/FLOAT qualifier
specifying floating-point format in DEC C, 12–5
Fortran
/CHECK qualifier, 2–18
qualifier needed for translated image support, 10–6
free routine
memory allocation, 6–1

G

Generating VAX instructions at run time, 2–5, 2–18, 2–24, 3–5, 3–7
$GETJPI system service
page-size dependencies, 6–4
$GETQUI system service
page-size dependencies, 6–4
$GETSYI system service, 2–14
determining host architecture, 5–4
obtaining the system page size, 6–20
page-size dependencies, 6–4
$GETUAI system service
page-size dependencies, 6–4
Global sections
alignment of, 2–5
creating, A–5
mapping, A–5
writable, 2–11
Global symbol tables
See GSTs
Granularity, 2–10, 2–12 to 2–14
byte, 2–12
of byte and word operations, 2–22, 2–24, 3–5
quadword, 2–12
specifying by compiler, 2–13
VEST qualifiers
memory, 2–24
GSTs (Global symbol tables), 2–25
G_floating data type, 1–3, 2–11

H

Heap Analyzer, 4–8
.H files
from SYS$STARLET_C.TLB to support DECthreads, 12–9
provided by SYS$STARLET_C.TLB, 12–9
HW_MODEL keyword
determining the host architecture, 5–5
H_floating data type, 1–3, 1–6, 2–7, 2–10, 2–19

I

IEEE data types
little endian, 1–2
IEEE floating-point data types, 2–11
specifying in DEC COBOL, 12–16
supported by DEC Ada, 12–2
supported by DEC C, 12–5
IIFs (image information files), 10–3
provided with Alpha software, 10–6, 10–8
Image information files
See IIFs
Images
creating, 5–2
translated
condition handling, 9–7
creating, 11–1
preserving atomicity in, 7–10
replacing with native Alpha image, 11–6
using in a link operation, 11–4
Imprecise exception reporting, 2–15
inadr argument
used with $CRETVA system service, 6–8
Include files
for C programs, 4–2
Initializing data structures
DEC C incompatibility with VAX C, 12–10
Inner access modes, 2–5, 2–6
INSQUEx instruction
accessing from DEC C, 12–6
Instructions
atomicity, 2–11 to 2–12
provided by PALcode, 1–6
VEST qualifiers, 2–23
memory barrier, 2–15
multiple instruction issue, 1–5
out-of-order completion, 1–6
parallel execution, 1–6
Instruction stream
inspecting, 2–5
Interlocked instructions
supported by DEC C, 12–7
Interoperability
between native and translated images, 10–2
compile-time considerations, 11–2
compiling native Alpha images, 11–1
confirming, 4–13
controlling the layout of symbol vectors, 11–6
creating native images that can be called by translated images, 11–5
creating native images that can call translated images, 11–2
creating stub images, 11–8
DEC COBOL support, 12–20
linking native Alpha images, 11–2
of native Alpha and translated images, 1–9, 2–20, 2–24
Interoperability (cont'd)
of translated and native images, 11–1
using the /BPAGE qualifier, 11–4
Interrupt priority level
See IPL
IPL (interrupt priority level)
elevated, 2–5
retained on Alpha, 1–6

J

Jacket routines, 2–25, 4–7
created automatically, 2–25
creating stub images, 11–8
writing for nonstandard calls, 2–25
JSB VAX instruction, 2–25

L

Languages, programming
See programming languages
$LCKPAG system service, A–5
page-size dependencies, 6–4
LIB$ESTABLISH routine, 2–17, 9–1, 12–8, 12–29, 12–37
support on Alpha systems, 9–12
LIB$FIND_IMAGE_SYMBOL routine, 10–5
LIB$FREE_VM_PAGE routine
page-size dependencies, 6–6
LIB$GET_VM_PAGE routine
page-size dependencies, 6–6
LIB$MATCH_COND routine, 9–6
LIB$REVERT routine, 2–17, 12–29
Librarian utility (LIBRARIAN)
native Alpha, 4–6
Library (LIB$) routines, 2–12
LIB$ESTABLISH, 2–17
LIB$REVERT, 2–17
not on Alpha, 1–2
Link commands
changes required, 4–4
Linker utility
/BPAGE option, 2–24
commands, 4–4
default page size, 4–4
features specific to OpenVMS Alpha, 5–2
native Alpha, 4–6
/NONATIVE_ONLY option, 2–25
options file changes, 1–1
Linking
creating native Alpha images, 5–2
creating native images that can call translated images, 11–2
Link procedures, 4–2
Little-endian data types, 1–2

$LKWSET system service, A–5
page-size dependencies, 6–4, 6–21
Load locked instruction (LDxL), 7–3
Load/store operations, 1–5
Locking mechanisms
for accessing byte variables, 2–13
Locking pages
page-size dependencies, 6–21
Locking services
$DEQ, 2–12, 2–15
$ENQ, 2–12, 2–15
Logical names
for tools and files, 4–2
run-time libraries, 10–10
systemwide definitions, 10–8

M

Machine instructions
creating, A–6
MACRO–32 compiler, 4–5
MACRO–64 assembler, 4–6
MACRO code
replacing, A–3
malloc routine
memory allocation, 6–1
Managing code migration, 1–8
Mapping memory
See Memory mapping
Mapping sections
into expanded virtual address space
page-size dependencies, 6–10
mapping a single page
page-size dependencies, 6–12
mapping into a defined address range
page-size dependencies, 6–13
page-size dependencies, 6–10
MAT functions used by translated BASIC images, 10–10
Mathematic routines
compatibility, 5–4
MB instruction
accessing from DEC C, 12–6
Mechanism array
format, 9–3
reliance on architectural details of, 2–17
using the depth argument, 9–5
/MEMBER_ALIGNMENT qualifier
controlling data alignment in DEC C, 12–7
Memory allocation
by expanding virtual address space
page-size dependencies, 6–6
finding page-size dependencies in, 6–6
freeing allocated memory
page-size dependencies, 6–9
page-size dependencies, 6–1
reallocating existing virtual addresses
Memory allocation
reallocating existing virtual addresses (cont'd)
page-size dependencies, 6–8
specifying address ranges, 6–8
specifying page counts, 6–6
using the $CRETVA system service, 6–9
using the $EXPREG system service, 6–7
Memory barrier
See MB instruction
Memory barrier instructions, 2–15
Memory locking
page-size dependencies, 6–1, 6–21
Memory management functions
page-size dependencies, 6–1
summary, 6–2 to 6–5
Memory-management system services, 2–14
Memory mapping
into expanded virtual address space
page-size dependencies, 6–10
mapping a single page
page-size dependencies, 6–12
mapping into a defined address range
page-size dependencies, 6–13
required changes, 6–16
page-size dependencies, 6–1, 6–10
using the $CRMPSC system service, 6–11
Memory protection
page-size dependencies, 6–1
page size granularity, 2–13
Message utility (MESSAGE)
native Alpha, 4–6
$MGBLSC system service, 2–14, A–5
page-size dependencies, 6–4
Migrating
ease of, 1–1
privileged code, 2–6
third-party products, 2–2
user-mode code, 1–1, 1–8
Migration Assessment Service, 1–10
Migration methods
and program architectural dependencies, 2–22
comparison of, 2–20
for user-mode code, 1–8
illustration of, 1–8
selecting, 2–3, 2–22
Migration planning
sample migration plan, 3–1
services, 1–10
Migration services
Application Migration, 1–10
Application Migration Detailed Analysis, 1–10
Migration Assessment, 1–10
System Migration, 1–10
System Migration Detailed Analysis, 1–10
Migration tools, 4–2

Migration training, 1–11
how to order, 1–11
Mixing native Alpha and translated images
as a stage in migration, 1–9
possibility of, 1–9
MMS (Module Management System), 2–2, 4–2
Module Management System
See MMS
MTH$ routines
compatibility, 5–4
MTH$ RTL
double-precision floating-point functions invoked by translated images, 10–10
translated, 10–10
Multiple instruction issue, 1–5
Multiprocessing, A–5

N

/NATIVE_ONLY qualifier, 11–4, 12–34
interoperability, 11–2
Natural alignment of data
See data alignment
Network interfaces
supported on Alpha, 1–3
/NOMEMBER_ALIGN qualifier
for DEC C compiler, 3–4
Nonstandard calls
writing jacket routines for, 2–25

O

OpenVMS Alpha operating system
compatibility goals of, 1–1
diagnostic features, 2–18
OpenVMS Mathematics Run-Time Library
compatibility, 5–4
Optimized code, 2–6
Optimizing compilers, 4–4
Order information
migration services, 1–10
migration training, 1–11
__OR_ATOMIC_LONG built-in, 12–7
__OR_ATOMIC_QUAD built-in, 12–7
OTS$CALL_PROC RTL routine
enabling callbacks to translated images, 11–1
Overflow detection
enabling, 9–12

P

Packed decimal data type, 2–10, 2–19
Pagelets
definition, 6–1
using with $EXPREG system service, 6–6
Page sizes, 1–5, 2–13 to 2–14, 3–5, A–5
compatibility with OpenVMS VAX, 6–1
dependencies on VAX page size, 6–1
hard-coded references, 2–14
memory protection granularity, 2–13, 2–24
permissive protection, 2–5, 2–22
relation to disk block size, 2–14
supported by Alpha systems, 6–1
using $GETSYI to obtain the page size at run time, 6–20
PALcode (privileged architecture library), 1–6
Parallel execution of instructions, 1–6
Parallel Processing Run-Time Library (PPL$) routines, 2–12, 2–15
PCA (Performance and Coverage Analyzer)
analyzing images, 2–20
detecting unaligned data, 2–10, 2–20
identifying critical images, 2–21
pcc
supported as DEC C compatibility mode, 12–4
PCs (Program counters), 2–5, 2–15
in signal array on Alpha systems, 9–3
modifying, 2–18
PDP–11 compatibility mode, 2–5
Performance
of translated images, 1–9, 10–1
Performance and Coverage Analyzer
See PCA
Performance monitors
non-Digital, 2–6
Permissive protection, 2–24
Planning a migration, 1–8, 2–1
Porting checklist, 2–7
#PRAGMA NO_MEMBER_ALIGNMENT, 2–10
Precise exception reporting, 2–15, 2–23, 2–24
VEST qualifiers, 2–24
/PRESERVE=FLOAT_EXCEPTIONS
translation qualifier needed for TIE condition handler, 10–4
Privileged architecture library
See PALcode
Privileged code
finding with VEST, 2–19
migrating to OpenVMS Alpha, 2–6
Privileged mode operation, A–6
Privileged shareable images, 2–6
Privileged VAX instructions, 2–5
Procedure arguments
accessing, 2–16
Procedure signature blocks
See PSBs
Processor modes
unchanged on Alpha, 1–6
Processor status longwords
See PSLs

Processor status word (PSW), 2–18
Process space
used by translated image, 2–5
Program counters
See PCs
Programming languages
See also specific languages; Compilers
Ada, 4–4
BASIC, 4–4
C, 4–4
include files, 4–2
/NOMEMBER_ALIGN qualifier, 3–4
VOLATILE declaration, 2–12
C++, 4–4
COBOL, 4–4
FORTRAN, 4–4
LISP, 4–4
Pascal, 4–4
PL/I, 4–4
VAX MACRO, 4–4
Protection
permissive, 2–24
PSBs (procedure signature blocks)
generating, 11–1
PSLs (Processor status longwords)
in signal array on Alpha systems, 9–3
$PURGWS system service
page-size dependencies, 6–5

Q

Quadword granularity, 2–12

R

Rdb/VMS
same function on Alpha, 1–3
Read/write operations
ordering of, 2–14 to 2–15, 2–23
Read/write ordering, 7–9
effect on synchronization, 7–3
Recompiling, 2–18
changes in compile commands, 4–4
comparison with translating, 2–20, 2–22
effect of architectural dependencies, 2–22 to 2–23
produces native Alpha image, 4–4
resolving errors, 4–4
restrictions, 2–3
to create native Alpha images, 1–8
Record Management Services
See RMS
Relinking, 4–6
changes in link commands, 4–4
to create native Alpha images, 1–8
REMQUEx instruction
accessing from DEC C, 12–6
retadr argument
used with $CRETVA system service, 6–9
used with $CRMPSC system service, 6–11
used with $EXPREG system service, 6–7
Return addresses
modifying on stack, 2–16
Reviewing application code, 2–19
RISC architecture
characteristics of, 1–5 to 1–6
RMS (Record Management Services)
unchanged for Alpha, 1–2
Rounding problem and workaround
in translated images, 10–4
Running translated images, 10–3
defining logical names for translated libraries, 10–3
Run-time library routines
accessing the D56 form, 10–10
calling interface unchanged, 1–2
different operation on Alpha, 1–2
LIB$ESTABLISH, 2–17
LIB$REVERT, 2–17
page-size dependencies, 6–6

S

SDA (System Dump Analyzer utility)
See System Dump Analyzer utility
Selecting a migration method, 2–3, 2–22
Self-modifying code, 2–5
$SETAST system service, 2–12
$SETPRT system service
page-size dependencies, 6–5
$SETUAI system service
page-size dependencies, 6–5
Shareable images
identifying, 2–1
linker options file changes required, 1–1
privileged, 2–6
replacing a translated image with a native image, 11–6
translated, 2–25
Shared data, 2–11
atomicity of, 2–12
unintentional sharing, 7–8
SIFs (symbol information files), 11–6
format, 11–7
Signal array
format, 9–2
reliance on architectural details of, 2–17
Sliced images, 4–11
$SNDJBC system service
page-size dependencies, 6–5

Software migration tools, 1–8
SS$_ALIGN exception, 9–7
signal array format, 9–10
SS$_HPARITH exception, 9–7
signal array format, 9–8
SS$_INVARG exception
mapping memory, 6–12
returned when mapping memory, 6–13
Stack
modifying return addresses on, 2–16
Stack switching, 2–5
Store conditional instruction (STxC), 7–3
String constants
modifying, 12–9
Stub images
creating, 11–8
Support for migration, 1–10
Switching stacks, 2–5
Symbols
redefining
DEC C incompatibility with VAX C, 12–10
Symbol vectors
controlling the layout of, 11–6
declaring universal symbols on Alpha systems, 5–2
SYMBOL_VECTOR= option
interoperability considerations, 11–6
Synchronization, 7–1 to 7–11
Alpha compatibility features, 7–3
and VEST, 2–20
checking for VAX assumptions, 7–3
example program, 7–5
explicit, 2–12
instructions, 2–23
latent problems, 2–19
of interprocess communication, A–5
of translated images, 7–10
using flag-passing protocols, 2–15
using system services, 2–15
VAX architectural features, 7–2
SYS$LIBRARY:LIB
compiling against, 2–6
SYS$STARLET_C.TLB
adherence to conventions, 12–9
functional equivalency to STARLETSD.TLB, 12–8
impact on use of "variant_struct" and "variant_union", 12–8
potential impact on LIB structures, 12–8
potential impact on RMS structures, 12–8
providing .H files, 12–9
SYS$UNWIND routine, 9–5
SYS.STB
linking against, 2–6, A–6
SYSGEN (System Generation utility)
See System Generation utility
SYSMAN (System Management utility)
See System Management utility
System-Code Debugger, 4–8
See also Debugger
OpenVMS Alpha, 4–10
System Dump Analyzer utility (SDA)
OpenVMS Alpha, 4–11
System dump files
analyzing, 4–11
System Generation utility (SYSGEN)
device configuration functions, 1–2
System information files
See SIFs
System library
compiling against, 2–6
System Management utility (SYSMAN)
device configuration functions, 1–2
System Migration Detailed Analysis Service, 1–10
System Migration Service, 1–10
System services
asynchronous, 2–12
calling interface unchanged, 1–2
$CMEXEC, 2–6
$CMKRNL, 2–6
$CRETVA, A–5
$CRMPSC, 2–6, 2–14, A–5
$DELTVA, A–5
$DEQ, 2–12, 2–15
different operation on Alpha, 1–2
$ENQ, 2–12, 2–15
$GETSYI, 2–14
$LCKPAG, A–5
$LKWSET, A–5
memory management, 2–14
memory management functions
page-size dependencies, 6–2
$MGBLSC, 2–14, A–5
protection problems created, A–5
replacing VAX MACRO code, 2–6
$SETAST, 2–12
undocumented, 2–5
$UPDSEC, A–5
user-written, 2–6
System space
reference to addresses in, 2–5, 2–6
System symbol table (SYS.STB)
linking against, 2–6
Systemwide logical names, 10–8

T

TESTBITCCI instruction
accessing from DEC C, 12–7
TESTBITSSI instruction
accessing from DEC C, 12–7

Text libraries
portability, 12–10
Third-party products
migrating, 2–2
Threaded code, 2–5
Threads of execution
effect on synchronization, 7–1
TIE$EMULAT_TV.EXE image, 10–5
TIE$SHARE shareable image, 10–2
TIE (Translated Image Environment), 1–2, 4–2, 10–2, 10–6
access violation workaround, 10–5
description, 4–7
interoperability between native and translated images, 10–2
invoked automatically, 4–7
restrictions, 10–4
running translated images, 10–3
statistics and feedback, 10–3
system logical names, 10–8
using /TIE qualifier to enable autojacketing, 10–5
/TIE qualifier
compiler interoperability qualifier, 11–1
DEC Fortran for OpenVMS Alpha support, 12–34
Training, 1–11
Translated VAX COBOL programs support, 10–12
Translated Image Environment
See TIE
Translated images
contents, 4–8
creating, 11–1
debugging, 4–10
description, 1–9
enabling callbacks to, 11–1
library routine calls, 1–2
performance of, 1–9, 10–1
preserving atomicity in, 7–10
system service calls, 1–2
using in a link operation, 11–4
Translated image support, 10–2, 10–6
See also TIE
additional qualifier required for FORTRAN, 10–6
need for additional steps, 10–6
Translated VAX C Run-Time Library, 10–11
functional restrictions, 10–11
interoperability restrictions, 10–12
Translating, 1–2, 4–6
See also VEST
as a stage in migration, 2–23
comparison with recompiling, 2–20, 2–22
effect of architectural dependencies, 2–22 to 2–23
for compatibility, 1–8, 2–23
performance of translated image, 1–9
Translating (cont'd)
programs in languages with no Alpha compiler, 4–4
restrictions, 2–3
tools for, 4–7
type of image produced, 4–8
Translation
BASIC images, 10–10
BLAS$, 10–10
callers to CRF$FREE_VM or CRF$GET_VM, 10–11
executable files, 10–6
images, 10–6
MTHRTL, 10–10
run-time libraries, 10–10
TRAPB instruction
accessing in DEC C, 12–6

U

$ULKPAG system service
page-size dependencies, 6–5
$ULWSET system service
page-size dependencies, 6–5
Unaligned data
cause of reduced performance, 1–9
in dynamic structures, 2–19
reduced performance, 10–1
supported under translation, 2–22
Unaligned variables, 2–19
Uninitialized variables, 2–19
Unwinding in exception handlers, 9–5
$UPDSEC system service, A–5
page-size dependencies, 6–5
User-mode images
slicing, 4–11
User-written device drivers
on OpenVMS Alpha systems, 1–6

V

Variables
shared
atomicity of, 2–12
unaligned, 2–19
uninitialized, 2–19
"variant_struct"
impact of SYS$STARLET_C.TLB, 12–8
"variant_union"
impact of SYS$STARLET_C.TLB, 12–8
VAX Ada
See DEC Ada
VAX architecture
dependencies, 2–9
general description, 1–4
VAX C
See DEC C

VAX calling standard
call frame stack, 3–5
reliance on, 2–16
VAXCDEF.TLB
replaced by new files, 12–8
VAX COBOL
See DEC COBOL
VAX dependency checklist, 2–7
VAX Environment Software Translator
See VEST
VAX FORTRAN
See DEC Fortran for OpenVMS VAX Systems
VAX instructions
accessing from DEC C, 12–6
CALL*x*, 2–25
generating at run time, 2–5, 2–18, 2–24, 3–5, 3–7
interlocked instructions
supported by DEC C, 12–7
interpreting, 4–7
JSB, 2–25
LONGJMP, 3–5
modifying, 2–18
privileged instructions, 2–5
reduced performance, 10–1
reliance on behavior of, 2–18
cause of reduced performance, 1–9
SETJMP, 3–5
supported in PALcode, 1–6
vector instructions, 2–5
VAX MACRO
See also MACRO–32 compiler
as compiled language, 2–6
LIB$ESTABLISH routine, 9–1
only a migration aid, 2–6
recompiling on OpenVMS Alpha systems, 4–5
replaced by system services, 2–6
VAX MACRO-32 compiler, 2–16
only a migration aid, 2–6
VAX Pascal
See DEC Pascal
VAXscan compiler, 2–3
Vector instructions, 2–5
VEST, 10–2
See also DECmigrate, Translated Image Environment (TIE), and Translated image support
VEST (VAX Environment Software Translator), 1–8, 4–2, 11–4
analytical ability, 4–8
and page size, 2–24
as analysis tool, 2–19
restrictions, 2–19
capabilities, 4–7
creating stub images, 11–8
/FLOAT=D53_FLOAT qualifier, 2–22
/FLOAT=D56_FLOAT qualifier, 2–22
VEST (VAX Environment Software Translator) (cont'd)
generating VAX instructions, 2–24
interoperability, 11–1
/OPTIMIZE=ALIGNMENT qualifier, 2–22, 2–23
/OPTIMIZE=NOALIGNMENT qualifier, 2–23
/PRESERVE=FLOAT_EXCEPTIONS qualifier, 2–23, 2–24
/PRESERVE=INSTRUCTION_ATOMICITY qualifier, 2–22, 2–23
/PRESERVE=INTEGER_EXCEPTIONS qualifier, 2–23, 2–24
/PRESERVE=MEMORY_ATOMICITY qualifier, 2–22, 2–24
/PRESERVE=READ_WRITE_ORDERING qualifier, 2–23
/PRESERVE qualifier, 7–10, 9–10
resources required, 4–2
runs on VAX and Alpha systems, 4–2
using symbol information files (SIF), 11–6
warning messages, 3–4
VEST/DEPENDENCY analysis tool, 2–1, 4–2
Virtual addresses
manipulating, A–5
Volatile attribute
protecting shared data, 7–3, 7–9
supported by DEC C, 12–8

W

Working set
modifying, A–5
Writable global sections, 2–11

NOTES

NOTES

NOTES

NOTES